```
**********************************************************
    1303 Mon Jul  9 14:38:05 2012
new/usr/src/cmd/cmd-inet/etc/sock2path.d/system%2Fkernel
dccp: lint fixes, dccp_conn_create_v6
**********************************************************
```

```
  1 # CDDL HEADER START
  2 #
  3 # The contents of this file are subject to the terms of the
  4 # Common Development and Distribution License (the "License").
  5 # You may not use this file except in compliance with the License.
  6 #
  7 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  8 # or http://www.opensolaris.org/os/licensing.
  9 # See the License for the specific language governing permissions
 10 # and limitations under the License.
 11 #
 12 # When distributing Covered Code, include this CDDL HEADER in each
 13 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 14 # If applicable, add the following below this CDDL HEADER, with the
 15 # fields enclosed by brackets "[]" replaced with your own identifying
 16 # information: Portions Copyright [yyyy] [name of copyright owner]
 17 #
 18 # CDDL HEADER END
 19 #
 20 # Copyright (c) 1995, 2010, Oracle and/or its affiliates. All rights reserved.
 21 #
 22 # socket configuration information
 23 #
 24 #     Family   Type  Protocol   Dev|Module
 25         2       2       0       tcp
 26         2       2       6       tcp

 28         26      2       0       tcp
 29         26      2       6       tcp

 31         2       1       0       udp
 32         2       1       17      udp

 34         26      1       0       udp
 35         26      1       17      udp

 37         1       2       0       /dev/ticotsord
 38         1       6       0       /dev/ticotsord
 39         1       1       0       /dev/ticlts

 41         2       4       0       icmp
 42         26      4       0       icmp

 44         2       2       132     socksctp
 45         26      2       132     socksctp
 46         2       6       132     socksctp
 47         26      6       132     socksctp

 49         24      4       0       rts

 51         27      4       2       /dev/keysock
 52         29      4       1       /dev/spdsock

 54         31      1       0       trill

 56         2       6       33      dccp
 57 #endif /* ! codereview */
```

```
**********************************************************
  189148 Mon Jul  9 14:38:05 2012
new/usr/src/cmd/cmd-inet/usr.bin/netstat/netstat.c
dccp: complete netstack
**********************************************************
______unchanged_portion_omitted_

 138 static  mib_item_t      *mibget(int sd);
 139 static  void            mibfree(mib_item_t *firstitem);
 140 static  int             mibopen(void);
 141 static void             mib_get_constants(mib_item_t *item);
 142 static mib_item_t       *mib_item_dup(mib_item_t *item);
 143 static mib_item_t       *mib_item_diff(mib_item_t *item1,
 144     mib_item_t *item2);
 145 static void             mib_item_destroy(mib_item_t **item);

 147 static boolean_t        octetstrmatch(const Octet_t *a, const Octet_t *b);
 148 static char             *octetstr(const Octet_t *op, int code,
 149                             char *dst, uint_t dstlen);
 150 static char             *pr_addr(uint_t addr,
 151                             char *dst, uint_t dstlen);
 152 static char             *pr_addrnz(ipaddr_t addr, char *dst, uint_t dstlen);
 153 static char             *pr_addr6(const in6_addr_t *addr,
 154                             char *dst, uint_t dstlen);
 155 static char             *pr_mask(uint_t addr,
 156                             char *dst, uint_t dstlen);
 157 static char             *pr_prefix6(const struct in6_addr *addr,
 158                             uint_t prefixlen, char *dst, uint_t dstlen);
 159 static char             *pr_ap(uint_t addr, uint_t port,
 160                             char *proto, char *dst, uint_t dstlen);
 161 static char             *pr_ap6(const in6_addr_t *addr, uint_t port,
 162                             char *proto, char *dst, uint_t dstlen);
 163 static char             *pr_net(uint_t addr, uint_t mask,
 164                             char *dst, uint_t dstlen);
 165 static char             *pr_netaddr(uint_t addr, uint_t mask,
 166                             char *dst, uint_t dstlen);
 167 static char             *fmodestr(uint_t fmode);
 168 static char             *portname(uint_t port, char *proto,
 169                             char *dst, uint_t dstlen);

 171 static const char       *mitcp_state(int code,
 172                             const mib2_transportMLPEntry_t *attr);
 173 static const char       *miudp_state(int code,
 174                             const mib2_transportMLPEntry_t *attr);

 176 static void             stat_report(mib_item_t *item);
 177 static void             mrt_stat_report(mib_item_t *item);
 178 static void             arp_report(mib_item_t *item);
 179 static void             ndp_report(mib_item_t *item);
 180 static void             mrt_report(mib_item_t *item);
 181 static void             if_stat_total(struct ifstat *oldstats,
 182                             struct ifstat *newstats, struct ifstat *sumstats);
 183 static void             if_report(mib_item_t *item, char *ifname,
 184                             int Iflag_only, boolean_t once_only);
 185 static void             if_report_ip4(mib2_ipAddrEntry_t *ap,
 186                             char ifname[], char logintname[],
 187                             struct ifstat *statptr, boolean_t ksp_not_null);
 188 static void             if_report_ip6(mib2_ipv6AddrEntry_t *ap6,
 189                             char ifname[], char logintname[],
 190                             struct ifstat *statptr, boolean_t ksp_not_null);
 191 static void             ire_report(const mib_item_t *item);
 192 static void             tcp_report(const mib_item_t *item);
 193 static void             udp_report(const mib_item_t *item);
 194 static void             group_report(mib_item_t *item);
 195 static void             dce_report(mib_item_t *item);
 196 static void             sctp_report(const mib_item_t *item);
```



```
 197 static void             dccp_report(const mib_item_t *item);
 198 #endif /* ! codereview */
 199 static void             print_ip_stats(mib2_ip_t *ip);
 200 static void             print_icmp_stats(mib2_icmp_t *icmp);
 201 static void             print_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6);
 202 static void             print_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6);
 203 static void             print_sctp_stats(mib2_sctp_t *tcp);
 204 static void             print_tcp_stats(mib2_tcp_t *tcp);
 205 static void             print_udp_stats(mib2_udp_t *udp);
 206 static void             print_rawip_stats(mib2_rawip_t *rawip);
 207 static void             print_igmp_stats(struct igmpstat *igps);
 208 static void             print_mrt_stats(struct mrtstat *mrts);
 209 static void             print_dccp_stats(mib2_dccp_t *dccp);
 196 static void             sctp_report(const mib_item_t *item);
 210 static void             sum_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6,
 211                             mib2_ipv6IfStatsEntry_t *sum6);
 212 static void             sum_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6,
 213                             mib2_ipv6IfIcmpEntry_t *sum6);
 214 static void             m_report(void);
 215 static void             dhcp_report(char *);

 217 static  uint64_t        kstat_named_value(kstat_t *, char *);
 218 static  kid_t           safe_kstat_read(kstat_ctl_t *, kstat_t *, void *);
 219 static int              isnum(char *);
 220 static char             *plural(int n);
 221 static char             *pluraly(int n);
 222 static char             *plurales(int n);
 223 static void             process_filter(char *arg);
 224 static char             *ifindex2str(uint_t, char *);
 225 static boolean_t        family_selected(int family);

 227 static void             usage(char *);
 228 static void             fatal(int errcode, char *str1, ...);

 230 #define PLURAL(n) plural((int)n)
 231 #define PLURALY(n) pluraly((int)n)
 232 #define PLURALES(n) plurales((int)n)
 233 #define IFLAGMOD(flg, val1, val2)       if (flg == val1) flg = val2
 234 #define MDIFF(diff, elem2, elem1, member)       (diff)->member = \
 235         (elem2)->member - (elem1)->member


 238 static  boolean_t       Aflag = B_FALSE;        /* All sockets/ifs/rtng-tbls */
 239 static  boolean_t       Dflag = B_FALSE;        /* DCE info */
 240 static  boolean_t       Iflag = B_FALSE;        /* IP Traffic Interfaces */
 241 static  boolean_t       Mflag = B_FALSE;        /* STREAMS Memory Statistics */
 242 static  boolean_t       Nflag = B_FALSE;        /* Numeric Network Addresses */
 243 static  boolean_t       Rflag = B_FALSE;        /* Routing Tables */
 244 static  boolean_t       RSECflag = B_FALSE;     /* Security attributes */
 245 static  boolean_t       Sflag = B_FALSE;        /* Per-protocol Statistics */
 246 static  boolean_t       Vflag = B_FALSE;        /* Verbose */
 247 static  boolean_t       Pflag = B_FALSE;        /* Net to Media Tables */
 248 static  boolean_t       Gflag = B_FALSE;        /* Multicast group membership */
 249 static  boolean_t       MMflag = B_FALSE;       /* Multicast routing table */
 250 static  boolean_t       DHCPflag = B_FALSE;     /* DHCP statistics */
 251 static  boolean_t       Xflag = B_FALSE;        /* Debug Info */

 253 static  int     v4compat = 0;   /* Compatible printing format for status */

 255 static int      proto = IPPROTO_MAX;    /* all protocols */
 256 kstat_ctl_t     *kc = NULL;

 258 /*
 259  * Sizes of data structures extracted from the base mib.
 260  * This allows the size of the tables entries to grow while preserving
 261  * binary compatibility.
```

```
 */
static int ipAddrEntrySize;
static int ipRouteEntrySize;
static int ipNetToMediaEntrySize;
static int ipMemberEntrySize;
static int ipGroupSourceEntrySize;
static int ipRouteAttributeSize;
static int vifctlSize;
static int mfcctlSize;

static int ipv6IfStatsEntrySize;
static int ipv6IfIcmpEntrySize;
static int ipv6AddrEntrySize;
static int ipv6RouteEntrySize;
static int ipv6NetToMediaEntrySize;
static int ipv6MemberEntrySize;
static int ipv6GroupSourceEntrySize;

static int ipDestEntrySize;

static int transportMLPSize;
static int tcpConnEntrySize;
static int tcp6ConnEntrySize;
static int udpEntrySize;
static int udp6EntrySize;
static int sctpEntrySize;
static int sctpLocalEntrySize;
static int sctpRemoteEntrySize;
static int dccpEntrySize;
static int dccp6EntrySize;
#endif /* ! codereview */

#define protocol_selected(p)    (proto == IPPROTO_MAX || proto == (p))

/* Machinery used for -f (filter) option */
enum { FK_AF = 0, FK_OUTIF, FK_DST, FK_FLAGS, NFILTERKEYS };

static const char *filter_keys[NFILTERKEYS] = {
        "af", "outif", "dst", "flags"
};

static m_label_t *zone_security_label = NULL;

/* Flags on routes */
#define FLF_A           0x00000001
#define FLF_b           0x00000002
#define FLF_D           0x00000004
#define FLF_G           0x00000008
#define FLF_H           0x00000010
#define FLF_L           0x00000020
#define FLF_U           0x00000040
#define FLF_M           0x00000080
#define FLF_S           0x00000100
#define FLF_C           0x00000200      /* IRE_IF_CLONE */
#define FLF_I           0x00000400      /* RTF_INDIRECT */
#define FLF_R           0x00000800      /* RTF_REJECT */
#define FLF_B           0x00001000      /* RTF_BLACKHOLE */
#define FLF_Z           0x00100000      /* RTF_ZONE */

static const char flag_list[] = "AbDGHLUMSCIRBZ";

typedef struct filter_rule filter_t;

struct filter_rule {
        filter_t *f_next;
        union {
```



```
                int f_family;
                const char *f_ifname;
                struct {
                        struct hostent *f_address;
                        in6_addr_t f_mask;
                } a;
                struct {
                        uint_t f_flagset;
                        uint_t f_flagclear;
                } f;
        } u;
};

/*
 * The user-specified filters are linked into lists separated by
 * keyword (type of filter).  Thus, the matching algorithm is:
 *      For each non-empty filter list
 *              If no filters in the list match
 *                      then stop here; route doesn't match
 *      If loop above completes, then route does match and will be
 *      displayed.
 */
static filter_t *filters[NFILTERKEYS];

static uint_t timestamp_fmt = NODATE;

#if !defined(TEXT_DOMAIN)               /* Should be defined by cc -D */
#define TEXT_DOMAIN "SYS_TEST"          /* Use this only if it isn't */
#endif

int
main(int argc, char **argv)
{
        char            *name;
        mib_item_t      *item = NULL;
        mib_item_t      *previtem = NULL;
        int             sd = -1;
        char    *ifname = NULL;
        int     interval = 0;   /* Single time by default */
        int     count = -1;     /* Forever */
        int     c;
        int     d;
        /*
         * Possible values of 'Iflag_only':
         * -1, no feature-flags;
         *  0, IFlag and other feature-flags enabled
         *  1, IFlag is the only feature-flag enabled
         * : trinary variable, modified using IFLAGMOD()
         */
        int Iflag_only = -1;
        boolean_t once_only = B_FALSE; /* '-i' with count > 1 */
        extern char     *optarg;
        extern int      optind;
        char *default_ip_str = NULL;

        name = argv[0];

        v4compat = get_compat_flag(&default_ip_str);
        if (v4compat == DEFAULT_PROT_BAD_VALUE)
                fatal(2, "%s: %s: Bad value for %s in %s\n", name,
                    default_ip_str, DEFAULT_IP, INET_DEFAULT_FILE);
        free(default_ip_str);

        (void) setlocale(LC_ALL, "");
        (void) textdomain(TEXT_DOMAIN);
```

```
394         while ((c = getopt(argc, argv, "adimnrspMgvxf:P:I:DRT:")) != -1) {
395                 switch ((char)c) {
396                 case 'a':               /* all connections */
397                         Aflag = B_TRUE;
398                         break;

400                 case 'd':               /* DCE info */
401                         Dflag = B_TRUE;
402                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
403                         break;

405                 case 'i':               /* interface (ill/ipif report) */
406                         Iflag = B_TRUE;
407                         IFLAGMOD(Iflag_only, -1, 1); /* '-i' exists */
408                         break;

410                 case 'm':               /* streams msg report */
411                         Mflag = B_TRUE;
412                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
413                         break;

415                 case 'n':               /* numeric format */
416                         Nflag = B_TRUE;
417                         break;

419                 case 'r':               /* route tables */
420                         Rflag = B_TRUE;
421                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
422                         break;

424                 case 'R':               /* security attributes */
425                         RSECflag = B_TRUE;
426                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
427                         break;

429                 case 's':               /* per-protocol statistics */
430                         Sflag = B_TRUE;
431                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
432                         break;

434                 case 'p':               /* arp/ndp table */
435                         Pflag = B_TRUE;
436                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
437                         break;

439                 case 'M':               /* multicast routing tables */
440                         MMflag = B_TRUE;
441                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
442                         break;

444                 case 'g':               /* multicast group membership */
445                         Gflag = B_TRUE;
446                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
447                         break;

449                 case 'v':               /* verbose output format */
450                         Vflag = B_TRUE;
451                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
452                         break;

454                 case 'x':               /* turn on debugging */
455                         Xflag = B_TRUE;
456                         break;

458                 case 'f':
459                         process_filter(optarg);
```



```
460                         break;

462                 case 'P':
463                         if (strcmp(optarg, "ip") == 0) {
464                                 proto = IPPROTO_IP;
465                         } else if (strcmp(optarg, "ipv6") == 0 ||
466                             strcmp(optarg, "ip6") == 0) {
467                                 v4compat = 0;   /* Overridden */
468                                 proto = IPPROTO_IPV6;
469                         } else if (strcmp(optarg, "icmp") == 0) {
470                                 proto = IPPROTO_ICMP;
471                         } else if (strcmp(optarg, "icmpv6") == 0 ||
472                             strcmp(optarg, "icmp6") == 0) {
473                                 v4compat = 0;   /* Overridden */
474                                 proto = IPPROTO_ICMPV6;
475                         } else if (strcmp(optarg, "igmp") == 0) {
476                                 proto = IPPROTO_IGMP;
477                         } else if (strcmp(optarg, "udp") == 0) {
478                                 proto = IPPROTO_UDP;
479                         } else if (strcmp(optarg, "tcp") == 0) {
480                                 proto = IPPROTO_TCP;
481                         } else if (strcmp(optarg, "sctp") == 0) {
482                                 proto = IPPROTO_SCTP;
483                         } else if (strcmp(optarg, "raw") == 0 ||
484                             strcmp(optarg, "rawip") == 0) {
485                                 proto = IPPROTO_RAW;
486                         } else if (strcmp(optarg, "dccp") == 0) {
487                                 proto = IPPROTO_DCCP;
488 #endif /* ! codereview */
489                         } else {
490                                 fatal(1, "%s: unknown protocol.\n", optarg);
491                         }
492                         break;

494                 case 'I':
495                         ifname = optarg;
496                         Iflag = B_TRUE;
497                         IFLAGMOD(Iflag_only, -1, 1); /* see macro def'n */
498                         break;

500                 case 'D':
501                         DHCPflag = B_TRUE;
502                         Iflag_only = 0;
503                         break;

505                 case 'T':
506                         if (optarg) {
507                                 if (*optarg == 'u')
508                                         timestamp_fmt = UDATE;
509                                 else if (*optarg == 'd')
510                                         timestamp_fmt = DDATE;
511                                 else
512                                         usage(name);
513                         } else {
514                                 usage(name);
515                         }
516                         break;

518                 case '?':
519                 default:
520                         usage(name);
521                 }
522         }

524         /*
525          * Make sure -R option is set only on a labeled system.
```

```
526          */
527         if (RSECflag && !is_system_labeled()) {
528                 (void) fprintf(stderr, "-R set but labeling is not enabled\n");
529                 usage(name);
530         }

532         /*
533          * Handle other arguments: find interval, count; the
534          * flags that accept 'interval' and 'count' are OR'd
535          * in the outermost 'if'; more flags may be added as
536          * required
537          */
538         if (Iflag || Sflag || Mflag) {
539                 for (d = optind; d < argc; d++) {
540                         if (isnum(argv[d])) {
541                                 interval = atoi(argv[d]);
542                                 if (d + 1 < argc &&
543                                     isnum(argv[d + 1])) {
544                                         count = atoi(argv[d + 1]);
545                                         optind++;
546                                 }
547                                 optind++;
548                                 if (interval == 0 || count == 0)
549                                         usage(name);
550                                 break;
551                         }
552                 }
553         }
554         if (optind < argc) {
555                 if (Iflag && isnum(argv[optind])) {
556                         count = atoi(argv[optind]);
557                         if (count == 0)
558                                 usage(name);
559                         optind++;
560                 }
561         }
562         if (optind < argc) {
563                 (void) fprintf(stderr,
564                     "%s: extra arguments\n", name);
565                 usage(name);
566         }
567         if (interval)
568                 setbuf(stdout, NULL);

570         if (DHCPflag) {
571                 dhcp_report(Iflag ? ifname : NULL);
572                 exit(0);
573         }

575         /*
576          * Get this process's security label if the -R switch is set.
577          * We use this label as the current zone's security label.
578          */
579         if (RSECflag) {
580                 zone_security_label = m_label_alloc(MAC_LABEL);
581                 if (zone_security_label == NULL)
582                         fatal(errno, "m_label_alloc() failed");
583                 if (getplabel(zone_security_label) < 0)
584                         fatal(errno, "getplabel() failed");
585         }

587         /* Get data structures: priming before iteration */
588         if (family_selected(AF_INET) || family_selected(AF_INET6)) {
589                 sd = mibopen();
590                 if (sd == -1)
591                         fatal(1, "can't open mib stream\n");
```



```
592                 if ((item = mibget(sd)) == NULL) {
593                         (void) close(sd);
594                         fatal(1, "mibget() failed\n");
595                 }
596                 /* Extract constant sizes - need do once only */
597                 mib_get_constants(item);
598         }
599         if ((kc = kstat_open()) == NULL) {
600                 mibfree(item);
601                 (void) close(sd);
602                 fail(1, "kstat_open(): can't open /dev/kstat");
603         }

605         if (interval <= 0) {
606                 count = 1;
607                 once_only = B_TRUE;
608         }
609         /* 'for' loop 1: */
610         for (;;) {
611                 mib_item_t *curritem = NULL; /* only for -[M]s */

613                 if (timestamp_fmt != NODATE)
614                         print_timestamp(timestamp_fmt);

616                 /* netstat: AF_INET[6] behaviour */
617                 if (family_selected(AF_INET) || family_selected(AF_INET6)) {
618                         if (Sflag) {
619                                 curritem = mib_item_diff(previtem, item);
620                                 if (curritem == NULL)
621                                         fatal(1, "can't process mib data, "
622                                             "out of memory\n");
623                                 mib_item_destroy(&previtem);
624                         }

626                         if (!(Dflag || Iflag || Rflag || Sflag || Mflag ||
627                             MMflag || Pflag || Gflag || DHCPflag)) {
628                                 if (protocol_selected(IPPROTO_UDP))
629                                         udp_report(item);
630                                 if (protocol_selected(IPPROTO_TCP))
631                                         tcp_report(item);
632                                 if (protocol_selected(IPPROTO_SCTP))
633                                         sctp_report(item);
634                                 if (protocol_selected(IPPROTO_DCCP))
635                                         dccp_report(item);
636 #endif /* ! codereview */
637                         }
638                         if (Iflag)
639                                 if_report(item, ifname, Iflag_only, once_only);
640                         if (Mflag)
641                                 m_report();
642                         if (Rflag)
643                                 ire_report(item);
644                         if (Sflag && MMflag) {
645                                 mrt_stat_report(curritem);
646                         } else {
647                                 if (Sflag)
648                                         stat_report(curritem);
649                                 if (MMflag)
650                                         mrt_report(item);
651                         }
652                         if (Gflag)
653                                 group_report(item);
654                         if (Pflag) {
655                                 if (family_selected(AF_INET))
656                                         arp_report(item);
657                                 if (family_selected(AF_INET6))
```

```
658                                     ndp_report(item);
659                             }
660                             if (Dflag)
661                                     dce_report(item);
662                             mib_item_destroy(&curritem);
663                     }

665                     /* netstat: AF_UNIX behaviour */
666                     if (family_selected(AF_UNIX) &&
667                         (!(Dflag || Iflag || Rflag || Sflag || Mflag ||
668                         MMflag || Pflag || Gflag)))
669                             unixpr(kc);
670                     (void) kstat_close(kc);

672                     /* iteration handling code */
673                     if (count > 0 && --count == 0)
674                             break;
675                     (void) sleep(interval);

677                     /* re-populating of data structures */
678                     if (family_selected(AF_INET) || family_selected(AF_INET6)) {
679                             if (Sflag) {
680                                     /* previtem is a cut-down list */
681                                     previtem = mib_item_dup(item);
682                                     if (previtem == NULL)
683                                             fatal(1, "can't process mib data, "
684                                                 "out of memory\n");
685                             }
686                             mibfree(item);
687                             (void) close(sd);
688                             if ((sd = mibopen()) == -1)
689                                     fatal(1, "can't open mib stream anymore\n");
690                             if ((item = mibget(sd)) == NULL) {
691                                     (void) close(sd);
692                                     fatal(1, "mibget() failed\n");
693                             }
694                     }
695                     if ((kc = kstat_open()) == NULL)
696                             fail(1, "kstat_open(): can't open /dev/kstat");

698             } /* 'for' loop 1 ends */
699             mibfree(item);
700             (void) close(sd);
701             if (zone_security_label != NULL)
702                     m_label_free(zone_security_label);

704             return (0);
705     }


708     static int
709     isnum(char *p)
710     {
711             int     len;
712             int     i;

714             len = strlen(p);
715             for (i = 0; i < len; i++)
716                     if (!isdigit(p[i]))
717                             return (0);
718             return (1);
719     }


722     /* ------------------------------- MIBGET -------------------------------- */
```



```
724     static mib_item_t *
725     mibget(int sd)
726     {
727             /*
728              * buf is an automatic for this function, so the
729              * compiler has complete control over its alignment;
730              * it is assumed this alignment is satisfactory for
731              * it to be casted to certain other struct pointers
732              * here, such as struct T_optmgmt_ack * .
733              */
734             uintptr_t               buf[512 / sizeof (uintptr_t)];
735             int                     flags;
736             int                     i, j, getcode;
737             struct strbuf           ctlbuf, databuf;
738             struct T_optmgmt_req    *tor = (struct T_optmgmt_req *)buf;
739             struct T_optmgmt_ack    *toa = (struct T_optmgmt_ack *)buf;
740             struct T_error_ack      *tea = (struct T_error_ack *)buf;
741             struct opthdr           *req;
742             mib_item_t              *first_item = NULL;
743             mib_item_t              *last_item  = NULL;
744             mib_item_t              *temp;

746             tor->PRIM_type = T_SVR4_OPTMGMT_REQ;
747             tor->OPT_offset = sizeof (struct T_optmgmt_req);
748             tor->OPT_length = sizeof (struct opthdr);
749             tor->MGMT_flags = T_CURRENT;


752             /*
753              * Note: we use the special level value below so that IP will return
754              * us information concerning IRE_MARK_TESTHIDDEN routes.
755              */
756             req = (struct opthdr *)&tor[1];
757             req->level = EXPER_IP_AND_ALL_IRES;
758             req->name  = 0;
759             req->len   = 1;

761             ctlbuf.buf = (char *)buf;
762             ctlbuf.len = tor->OPT_length + tor->OPT_offset;
763             flags = 0;
764             if (putmsg(sd, &ctlbuf, (struct strbuf *)0, flags) == -1) {
765                     perror("mibget: putmsg(ctl) failed");
766                     goto error_exit;
767             }

769             /*
770              * Each reply consists of a ctl part for one fixed structure
771              * or table, as defined in mib2.h.  The format is a T_OPTMGMT_ACK,
772              * containing an opthdr structure.  level/name identify the entry,
773              * len is the size of the data part of the message.
774              */
775             req = (struct opthdr *)&toa[1];
776             ctlbuf.maxlen = sizeof (buf);
777             j = 1;
778             for (;;) {
779                     flags = 0;
780                     getcode = getmsg(sd, &ctlbuf, (struct strbuf *)0, &flags);
781                     if (getcode == -1) {
782                             perror("mibget getmsg(ctl) failed");
783                             if (Xflag) {
784                                     (void) fputs("#   level   name    len\n",
785                                         stderr);
786                                     i = 0;
787                                     for (last_item = first_item; last_item;
788                                         last_item = last_item->next_item)
789                                             (void) printf("%d  %4d   %5d   %d\n",
```

```
790                                     ++i,
791                                     last_item->group,
792                                     last_item->mib_id,
793                                     last_item->length);
794                     }
795                     goto error_exit;
796             }
797             if (getcode == 0 &&
798                 ctlbuf.len >= sizeof (struct T_optmgmt_ack) &&
799                 toa->PRIM_type == T_OPTMGMT_ACK &&
800                 toa->MGMT_flags == T_SUCCESS &&
801                 req->len == 0) {
802                     if (Xflag)
803                             (void) printf("mibget getmsg() %d returned "
804                                 "EOD (level %ld, name %ld)\n",
805                                 j, req->level, req->name);
806                     return (first_item);            /* this is EOD msg */
807             }

809             if (ctlbuf.len >= sizeof (struct T_error_ack) &&
810                 tea->PRIM_type == T_ERROR_ACK) {
811                     (void) fprintf(stderr,
812                         "mibget %d gives T_ERROR_ACK: TLI_error = 0x%lx, "
813                         "UNIX_error = 0x%lx\n",
814                         j, tea->TLI_error, tea->UNIX_error);

816                     errno = (tea->TLI_error == TSYSERR) ?
817                         tea->UNIX_error : EPROTO;
818                     goto error_exit;
819             }

821             if (getcode != MOREDATA ||
822                 ctlbuf.len < sizeof (struct T_optmgmt_ack) ||
823                 toa->PRIM_type != T_OPTMGMT_ACK ||
824                 toa->MGMT_flags != T_SUCCESS) {
825                     (void) printf("mibget getmsg(ctl) %d returned %d, "
826                         "ctlbuf.len = %d, PRIM_type = %ld\n",
827                         j, getcode, ctlbuf.len, toa->PRIM_type);

829                     if (toa->PRIM_type == T_OPTMGMT_ACK)
830                             (void) printf("T_OPTMGMT_ACK: "
831                                 "MGMT_flags = 0x%lx, req->len = %ld\n",
832                                 toa->MGMT_flags, req->len);
833                     errno = ENOMSG;
834                     goto error_exit;
835             }

837             temp = (mib_item_t *)malloc(sizeof (mib_item_t));
838             if (temp == NULL) {
839                     perror("mibget malloc failed");
840                     goto error_exit;
841             }
842             if (last_item != NULL)
843                     last_item->next_item = temp;
844             else
845                     first_item = temp;
846             last_item = temp;
847             last_item->next_item = NULL;
848             last_item->group = req->level;
849             last_item->mib_id = req->name;
850             last_item->length = req->len;
851             last_item->valp = malloc((int)req->len);
852             if (last_item->valp == NULL)
853                     goto error_exit;
854             if (Xflag)
855                     (void) printf("msg %d: group = %4d   mib_id = %5d"
```



```
856                             "length = %d\n",
857                             j, last_item->group, last_item->mib_id,
858                             last_item->length);

860                 databuf.maxlen = last_item->length;
861                 databuf.buf    = (char *)last_item->valp;
862                 databuf.len    = 0;
863                 flags = 0;
864                 getcode = getmsg(sd, (struct strbuf *)0, &databuf, &flags);
865                 if (getcode == -1) {
866                         perror("mibget getmsg(data) failed");
867                         goto error_exit;
868                 } else if (getcode != 0) {
869                         (void) printf("mibget getmsg(data) returned %d, "
870                             "databuf.maxlen = %d, databuf.len = %d\n",
871                             getcode, databuf.maxlen, databuf.len);
872                         goto error_exit;
873                 }
874                 j++;
875         }
876         /* NOTREACHED */

878 error_exit:;
879         mibfree(first_item);
880         return (NULL);
881 }

883 /*
884  * mibfree: frees a linked list of type (mib_item_t *)
885  * returned by mibget(); this is NOT THE SAME AS
886  * mib_item_destroy(), so should be used for objects
887  * returned by mibget() only
888  */
889 static void
890 mibfree(mib_item_t *firstitem)
891 {
892         mib_item_t *lastitem;

894         while (firstitem != NULL) {
895                 lastitem = firstitem;
896                 firstitem = firstitem->next_item;
897                 if (lastitem->valp != NULL)
898                         free(lastitem->valp);
899                 free(lastitem);
900         }
901 }

903 static int
904 mibopen(void)
905 {
906         int     sd;

908         sd = open("/dev/arp", O_RDWR);
909         if (sd == -1) {
910                 perror("arp open");
911                 return (-1);
912         }
913         if (ioctl(sd, I_PUSH, "tcp") == -1) {
914                 perror("tcp I_PUSH");
915                 (void) close(sd);
916                 return (-1);
917         }
918         if (ioctl(sd, I_PUSH, "udp") == -1) {
919                 perror("udp I_PUSH");
920                 (void) close(sd);
921                 return (-1);
```

```
922         }
923         if (ioctl(sd, I_PUSH, "icmp") == -1) {
924                 perror("icmp I_PUSH");
925                 (void) close(sd);
926                 return (-1);
927         }
928         return (sd);
929 }

931 /*
932  * mib_item_dup: returns a clean mib_item_t * linked
933  * list, so that for every element item->mib_id is 0;
934  * to deallocate this linked list, use mib_item_destroy
935  */
936 static mib_item_t *
937 mib_item_dup(mib_item_t *item)
938 {
939         int     c = 0;
940         mib_item_t *localp;
941         mib_item_t *tempp;

943         for (tempp = item; tempp; tempp = tempp->next_item)
944                 if (tempp->mib_id == 0)
945                         c++;
946         tempp = NULL;

948         localp = (mib_item_t *)malloc(c * sizeof (mib_item_t));
949         if (localp == NULL)
950                 return (NULL);
951         c = 0;
952         for (; item; item = item->next_item) {
953                 if (item->mib_id == 0) {
954                         /* Replicate item in localp */
955                         (localp[c]).next_item = NULL;
956                         (localp[c]).group = item->group;
957                         (localp[c]).mib_id = item->mib_id;
958                         (localp[c]).length = item->length;
959                         (localp[c]).valp = (uintptr_t *)malloc(
960                             item->length);
961                         if ((localp[c]).valp == NULL) {
962                                 mib_item_destroy(&localp);
963                                 return (NULL);
964                         }
965                         (void *) memcpy((localp[c]).valp,
966                             item->valp,
967                             item->length);
968                         tempp = &(localp[c]);
969                         if (c > 0)
970                                 (localp[c - 1]).next_item = tempp;
971                         c++;
972                 }
973         }
974         return (localp);
975 }

977 /*
978  * mib_item_diff: takes two (mib_item_t *) linked lists
979  * item1 and item2 and computes the difference between
980  * differentiable values in item2 against item1 for every
981  * given member of item2; returns an mib_item_t * linked
982  * list of diff's, or a copy of item2 if item1 is NULL;
983  * will return NULL if system out of memory; works only
984  * for item->mib_id == 0
985  */
986 static mib_item_t *
987 mib_item_diff(mib_item_t *item1, mib_item_t *item2) {
```



```
988         int     nitems  = 0; /* no. of items in item2 */
989         mib_item_t *tempp2;  /* walking copy of item2 */
990         mib_item_t *tempp1;  /* walking copy of item1 */
991         mib_item_t *diffp;
992         mib_item_t *diffptr; /* walking copy of diffp */
993         mib_item_t *prevp = NULL;

995         if (item1 == NULL) {
996                 diffp = mib_item_dup(item2);
997                 return (diffp);
998         }

1000        for (tempp2 = item2;
1001            tempp2;
1002            tempp2 = tempp2->next_item) {
1003                if (tempp2->mib_id == 0)
1004                        switch (tempp2->group) {
1005                        /*
1006                         * upon adding a case here, the same
1007                         * must also be added in the next
1008                         * switch statement, alongwith
1009                         * appropriate code
1010                         */
1011                        case MIB2_IP:
1012                        case MIB2_IP6:
1013                        case EXPER_DVMRP:
1014                        case EXPER_IGMP:
1015                        case MIB2_ICMP:
1016                        case MIB2_ICMP6:
1017                        case MIB2_TCP:
1018                        case MIB2_UDP:
1019                        case MIB2_SCTP:
1020                        case EXPER_RAWIP:
1021                        case MIB2_DCCP:
1022 #endif /* ! codereview */
1023                                nitems++;
1024                        }
1025        }
1026        tempp2 = NULL;
1027        if (nitems == 0) {
1028                diffp = mib_item_dup(item2);
1029                return (diffp);
1030        }

1032        diffp = (mib_item_t *)calloc(nitems, sizeof (mib_item_t));
1033        if (diffp == NULL)
1034                return (NULL);
1035        diffptr = diffp;
1036        /* 'for' loop 1: */
1037        for (tempp2 = item2; tempp2 != NULL; tempp2 = tempp2->next_item) {
1038                if (tempp2->mib_id != 0)
1039                        continue; /* 'for' loop 1 */
1040                /* 'for' loop 2: */
1041                for (tempp1 = item1; tempp1 != NULL;
1042                    tempp1 = tempp1->next_item) {
1043                        if (!(tempp1->mib_id == 0 &&
1044                            tempp1->group == tempp2->group &&
1045                            tempp1->mib_id == tempp2->mib_id))
1046                                continue; /* 'for' loop 2 */
1047                        /* found comparable data sets */
1048                        if (prevp != NULL)
1049                                prevp->next_item = diffptr;
1050                        switch (tempp2->group) {
1051                        /*
1052                         * Indenting note: Because of long variable names
1053                         * in cases MIB2_IP6 and MIB2_ICMP6, their contents
```

```
1054                         * have been indented by one tab space only
1055                         */
1056                        case MIB2_IP: {
1057                                mib2_ip_t *i2 = (mib2_ip_t *)tempp2->valp;
1058                                mib2_ip_t *i1 = (mib2_ip_t *)tempp1->valp;
1059                                mib2_ip_t *d;

1061                                diffptr->group = tempp2->group;
1062                                diffptr->mib_id = tempp2->mib_id;
1063                                diffptr->length = tempp2->length;
1064                                d = (mib2_ip_t *)calloc(tempp2->length, 1);
1065                                if (d == NULL)
1066                                        goto mibdiff_out_of_memory;
1067                                diffptr->valp = d;
1068                                d->ipForwarding = i2->ipForwarding;
1069                                d->ipDefaultTTL = i2->ipDefaultTTL;
1070                                MDIFF(d, i2, i1, ipInReceives);
1071                                MDIFF(d, i2, i1, ipInHdrErrors);
1072                                MDIFF(d, i2, i1, ipInAddrErrors);
1073                                MDIFF(d, i2, i1, ipInCksumErrs);
1074                                MDIFF(d, i2, i1, ipForwDatagrams);
1075                                MDIFF(d, i2, i1, ipForwProhibits);
1076                                MDIFF(d, i2, i1, ipInUnknownProtos);
1077                                MDIFF(d, i2, i1, ipInDiscards);
1078                                MDIFF(d, i2, i1, ipInDelivers);
1079                                MDIFF(d, i2, i1, ipOutRequests);
1080                                MDIFF(d, i2, i1, ipOutDiscards);
1081                                MDIFF(d, i2, i1, ipOutNoRoutes);
1082                                MDIFF(d, i2, i1, ipReasmTimeout);
1083                                MDIFF(d, i2, i1, ipReasmReqds);
1084                                MDIFF(d, i2, i1, ipReasmOKs);
1085                                MDIFF(d, i2, i1, ipReasmFails);
1086                                MDIFF(d, i2, i1, ipReasmDuplicates);
1087                                MDIFF(d, i2, i1, ipReasmPartDups);
1088                                MDIFF(d, i2, i1, ipFragOKs);
1089                                MDIFF(d, i2, i1, ipFragFails);
1090                                MDIFF(d, i2, i1, ipFragCreates);
1091                                MDIFF(d, i2, i1, ipRoutingDiscards);
1092                                MDIFF(d, i2, i1, tcpInErrs);
1093                                MDIFF(d, i2, i1, udpNoPorts);
1094                                MDIFF(d, i2, i1, udpInCksumErrs);
1095                                MDIFF(d, i2, i1, udpInOverflows);
1096                                MDIFF(d, i2, i1, rawipInOverflows);
1097                                MDIFF(d, i2, i1, ipsecInSucceeded);
1098                                MDIFF(d, i2, i1, ipsecInFailed);
1099                                MDIFF(d, i2, i1, ipInIPv6);
1100                                MDIFF(d, i2, i1, ipOutIPv6);
1101                                MDIFF(d, i2, i1, ipOutSwitchIPv6);
1102                                prevp = diffptr++;
1103                                break;
1104                        }
1105                        case MIB2_IP6: {
1106                        mib2_ipv6IfStatsEntry_t *i2;
1107                        mib2_ipv6IfStatsEntry_t *i1;
1108                        mib2_ipv6IfStatsEntry_t *d;

1110                        i2 = (mib2_ipv6IfStatsEntry_t *)tempp2->valp;
1111                        i1 = (mib2_ipv6IfStatsEntry_t *)tempp1->valp;
1112                        diffptr->group = tempp2->group;
1113                        diffptr->mib_id = tempp2->mib_id;
1114                        diffptr->length = tempp2->length;
1115                        d = (mib2_ipv6IfStatsEntry_t *)calloc(
1116                            tempp2->length, 1);
1117                        if (d == NULL)
1118                                goto mibdiff_out_of_memory;
1119                        diffptr->valp = d;
```



```
1120                        d->ipv6Forwarding = i2->ipv6Forwarding;
1121                        d->ipv6DefaultHopLimit =
1122                            i2->ipv6DefaultHopLimit;

1124                        MDIFF(d, i2, i1, ipv6InReceives);
1125                        MDIFF(d, i2, i1, ipv6InHdrErrors);
1126                        MDIFF(d, i2, i1, ipv6InTooBigErrors);
1127                        MDIFF(d, i2, i1, ipv6InNoRoutes);
1128                        MDIFF(d, i2, i1, ipv6InAddrErrors);
1129                        MDIFF(d, i2, i1, ipv6InUnknownProtos);
1130                        MDIFF(d, i2, i1, ipv6InTruncatedPkts);
1131                        MDIFF(d, i2, i1, ipv6InDiscards);
1132                        MDIFF(d, i2, i1, ipv6InDelivers);
1133                        MDIFF(d, i2, i1, ipv6OutForwDatagrams);
1134                        MDIFF(d, i2, i1, ipv6OutRequests);
1135                        MDIFF(d, i2, i1, ipv6OutDiscards);
1136                        MDIFF(d, i2, i1, ipv6OutNoRoutes);
1137                        MDIFF(d, i2, i1, ipv6OutFragOKs);
1138                        MDIFF(d, i2, i1, ipv6OutFragFails);
1139                        MDIFF(d, i2, i1, ipv6OutFragCreates);
1140                        MDIFF(d, i2, i1, ipv6ReasmReqds);
1141                        MDIFF(d, i2, i1, ipv6ReasmOKs);
1142                        MDIFF(d, i2, i1, ipv6ReasmFails);
1143                        MDIFF(d, i2, i1, ipv6InMcastPkts);
1144                        MDIFF(d, i2, i1, ipv6OutMcastPkts);
1145                        MDIFF(d, i2, i1, ipv6ReasmDuplicates);
1146                        MDIFF(d, i2, i1, ipv6ReasmPartDups);
1147                        MDIFF(d, i2, i1, ipv6ForwProhibits);
1148                        MDIFF(d, i2, i1, udpInCksumErrs);
1149                        MDIFF(d, i2, i1, udpInOverflows);
1150                        MDIFF(d, i2, i1, rawipInOverflows);
1151                        MDIFF(d, i2, i1, ipv6InIPv4);
1152                        MDIFF(d, i2, i1, ipv6OutIPv4);
1153                        MDIFF(d, i2, i1, ipv6OutSwitchIPv4);
1154                        prevp = diffptr++;
1155                        break;
1156                        }
1157                        case EXPER_DVMRP: {
1158                                struct mrtstat *m2;
1159                                struct mrtstat *m1;
1160                                struct mrtstat *d;

1162                                m2 = (struct mrtstat *)tempp2->valp;
1163                                m1 = (struct mrtstat *)tempp1->valp;
1164                                diffptr->group = tempp2->group;
1165                                diffptr->mib_id = tempp2->mib_id;
1166                                diffptr->length = tempp2->length;
1167                                d = (struct mrtstat *)calloc(tempp2->length, 1);
1168                                if (d == NULL)
1169                                        goto mibdiff_out_of_memory;
1170                                diffptr->valp = d;
1171                                MDIFF(d, m2, m1, mrts_mfc_hits);
1172                                MDIFF(d, m2, m1, mrts_mfc_misses);
1173                                MDIFF(d, m2, m1, mrts_fwd_in);
1174                                MDIFF(d, m2, m1, mrts_fwd_out);
1175                                d->mrts_upcalls = m2->mrts_upcalls;
1176                                MDIFF(d, m2, m1, mrts_fwd_drop);
1177                                MDIFF(d, m2, m1, mrts_bad_tunnel);
1178                                MDIFF(d, m2, m1, mrts_cant_tunnel);
1179                                MDIFF(d, m2, m1, mrts_wrong_if);
1180                                MDIFF(d, m2, m1, mrts_upq_ovflw);
1181                                MDIFF(d, m2, m1, mrts_cache_cleanups);
1182                                MDIFF(d, m2, m1, mrts_drop_sel);
1183                                MDIFF(d, m2, m1, mrts_q_overflow);
1184                                MDIFF(d, m2, m1, mrts_pkt2large);
1185                                MDIFF(d, m2, m1, mrts_pim_badversion);
```

```
1186                            MDIFF(d, m2, m1, mrts_pim_rcv_badcsum);
1187                            MDIFF(d, m2, m1, mrts_pim_badregisters);
1188                            MDIFF(d, m2, m1, mrts_pim_regforwards);
1189                            MDIFF(d, m2, m1, mrts_pim_regsend_drops);
1190                            MDIFF(d, m2, m1, mrts_pim_malformed);
1191                            MDIFF(d, m2, m1, mrts_pim_nomemory);
1192                            prevp = diffptr++;
1193                            break;
1194                    }
1195                    case EXPER_IGMP: {
1196                            struct igmpstat *i2;
1197                            struct igmpstat *i1;
1198                            struct igmpstat *d;
1199
1200                            i2 = (struct igmpstat *)tempp2->valp;
1201                            i1 = (struct igmpstat *)tempp1->valp;
1202                            diffptr->group = tempp2->group;
1203                            diffptr->mib_id = tempp2->mib_id;
1204                            diffptr->length = tempp2->length;
1205                            d = (struct igmpstat *)calloc(
1206                                tempp2->length, 1);
1207                            if (d == NULL)
1208                                    goto mibdiff_out_of_memory;
1209                            diffptr->valp = d;
1210                            MDIFF(d, i2, i1, igps_rcv_total);
1211                            MDIFF(d, i2, i1, igps_rcv_tooshort);
1212                            MDIFF(d, i2, i1, igps_rcv_badsum);
1213                            MDIFF(d, i2, i1, igps_rcv_queries);
1214                            MDIFF(d, i2, i1, igps_rcv_badqueries);
1215                            MDIFF(d, i2, i1, igps_rcv_reports);
1216                            MDIFF(d, i2, i1, igps_rcv_badreports);
1217                            MDIFF(d, i2, i1, igps_rcv_ourreports);
1218                            MDIFF(d, i2, i1, igps_snd_reports);
1219                            prevp = diffptr++;
1220                            break;
1221                    }
1222                    case MIB2_ICMP: {
1223                            mib2_icmp_t *i2;
1224                            mib2_icmp_t *i1;
1225                            mib2_icmp_t *d;
1226
1227                            i2 = (mib2_icmp_t *)tempp2->valp;
1228                            i1 = (mib2_icmp_t *)tempp1->valp;
1229                            diffptr->group = tempp2->group;
1230                            diffptr->mib_id = tempp2->mib_id;
1231                            diffptr->length = tempp2->length;
1232                            d = (mib2_icmp_t *)calloc(tempp2->length, 1);
1233                            if (d == NULL)
1234                                    goto mibdiff_out_of_memory;
1235                            diffptr->valp = d;
1236                            MDIFF(d, i2, i1, icmpInMsgs);
1237                            MDIFF(d, i2, i1, icmpInErrors);
1238                            MDIFF(d, i2, i1, icmpInCksumErrs);
1239                            MDIFF(d, i2, i1, icmpInUnknowns);
1240                            MDIFF(d, i2, i1, icmpInDestUnreachs);
1241                            MDIFF(d, i2, i1, icmpInTimeExcds);
1242                            MDIFF(d, i2, i1, icmpInParmProbs);
1243                            MDIFF(d, i2, i1, icmpInSrcQuenchs);
1244                            MDIFF(d, i2, i1, icmpInRedirects);
1245                            MDIFF(d, i2, i1, icmpInBadRedirects);
1246                            MDIFF(d, i2, i1, icmpInEchos);
1247                            MDIFF(d, i2, i1, icmpInEchoReps);
1248                            MDIFF(d, i2, i1, icmpInTimestamps);
1249                            MDIFF(d, i2, i1, icmpInAddrMasks);
1250                            MDIFF(d, i2, i1, icmpInAddrMaskReps);
1251                            MDIFF(d, i2, i1, icmpInFragNeeded);
```



```
1252                            MDIFF(d, i2, i1, icmpOutMsgs);
1253                            MDIFF(d, i2, i1, icmpOutDrops);
1254                            MDIFF(d, i2, i1, icmpOutErrors);
1255                            MDIFF(d, i2, i1, icmpOutDestUnreachs);
1256                            MDIFF(d, i2, i1, icmpOutTimeExcds);
1257                            MDIFF(d, i2, i1, icmpOutParmProbs);
1258                            MDIFF(d, i2, i1, icmpOutSrcQuenchs);
1259                            MDIFF(d, i2, i1, icmpOutRedirects);
1260                            MDIFF(d, i2, i1, icmpOutEchos);
1261                            MDIFF(d, i2, i1, icmpOutEchoReps);
1262                            MDIFF(d, i2, i1, icmpOutTimestamps);
1263                            MDIFF(d, i2, i1, icmpOutTimestampReps);
1264                            MDIFF(d, i2, i1, icmpOutAddrMasks);
1265                            MDIFF(d, i2, i1, icmpOutAddrMaskReps);
1266                            MDIFF(d, i2, i1, icmpOutFragNeeded);
1267                            MDIFF(d, i2, i1, icmpInOverflows);
1268                            prevp = diffptr++;
1269                            break;
1270                    }
1271                    case MIB2_ICMP6: {
1272            mib2_ipv6IfIcmpEntry_t *i2;
1273            mib2_ipv6IfIcmpEntry_t *i1;
1274            mib2_ipv6IfIcmpEntry_t *d;
1275
1276            i2 = (mib2_ipv6IfIcmpEntry_t *)tempp2->valp;
1277            i1 = (mib2_ipv6IfIcmpEntry_t *)tempp1->valp;
1278            diffptr->group = tempp2->group;
1279            diffptr->mib_id = tempp2->mib_id;
1280            diffptr->length = tempp2->length;
1281            d = (mib2_ipv6IfIcmpEntry_t *)calloc(tempp2->length, 1);
1282            if (d == NULL)
1283                    goto mibdiff_out_of_memory;
1284            diffptr->valp = d;
1285            MDIFF(d, i2, i1, ipv6IfIcmpInMsgs);
1286            MDIFF(d, i2, i1, ipv6IfIcmpInErrors);
1287            MDIFF(d, i2, i1, ipv6IfIcmpInDestUnreachs);
1288            MDIFF(d, i2, i1, ipv6IfIcmpInAdminProhibs);
1289            MDIFF(d, i2, i1, ipv6IfIcmpInTimeExcds);
1290            MDIFF(d, i2, i1, ipv6IfIcmpInParmProblems);
1291            MDIFF(d, i2, i1, ipv6IfIcmpInPktTooBigs);
1292            MDIFF(d, i2, i1, ipv6IfIcmpInEchos);
1293            MDIFF(d, i2, i1, ipv6IfIcmpInEchoReplies);
1294            MDIFF(d, i2, i1, ipv6IfIcmpInRouterSolicits);
1295            MDIFF(d, i2, i1, ipv6IfIcmpInRouterAdvertisements);
1296            MDIFF(d, i2, i1, ipv6IfIcmpInNeighborSolicits);
1297            MDIFF(d, i2, i1, ipv6IfIcmpInNeighborAdvertisements);
1298            MDIFF(d, i2, i1, ipv6IfIcmpInRedirects);
1299            MDIFF(d, i2, i1, ipv6IfIcmpInBadRedirects);
1300            MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembQueries);
1301            MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembResponses);
1302            MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembReductions);
1303            MDIFF(d, i2, i1, ipv6IfIcmpInOverflows);
1304            MDIFF(d, i2, i1, ipv6IfIcmpOutMsgs);
1305            MDIFF(d, i2, i1, ipv6IfIcmpOutErrors);
1306            MDIFF(d, i2, i1, ipv6IfIcmpOutDestUnreachs);
1307            MDIFF(d, i2, i1, ipv6IfIcmpOutAdminProhibs);
1308            MDIFF(d, i2, i1, ipv6IfIcmpOutTimeExcds);
1309            MDIFF(d, i2, i1, ipv6IfIcmpOutParmProblems);
1310            MDIFF(d, i2, i1, ipv6IfIcmpOutPktTooBigs);
1311            MDIFF(d, i2, i1, ipv6IfIcmpOutEchos);
1312            MDIFF(d, i2, i1, ipv6IfIcmpOutEchoReplies);
1313            MDIFF(d, i2, i1, ipv6IfIcmpOutRouterSolicits);
1314            MDIFF(d, i2, i1, ipv6IfIcmpOutRouterAdvertisements);
1315            MDIFF(d, i2, i1, ipv6IfIcmpOutNeighborSolicits);
1316            MDIFF(d, i2, i1, ipv6IfIcmpOutNeighborAdvertisements);
1317            MDIFF(d, i2, i1, ipv6IfIcmpOutRedirects);
```

```
1318         MDIFF(d, i2, i1, ipv6IfIcmpOutGroupMembQueries);
1319         MDIFF(d, i2, i1, ipv6IfIcmpOutGroupMembResponses);
1320         MDIFF(d, i2, i1, ipv6IfIcmpOutGroupMembReductions);
1321         prevp = diffptr++;
1322         break;
1323                         }
1324                         case MIB2_TCP: {
1325                                 mib2_tcp_t *t2;
1326                                 mib2_tcp_t *t1;
1327                                 mib2_tcp_t *d;

1329                                 t2 = (mib2_tcp_t *)tempp2->valp;
1330                                 t1 = (mib2_tcp_t *)tempp1->valp;
1331                                 diffptr->group = tempp2->group;
1332                                 diffptr->mib_id = tempp2->mib_id;
1333                                 diffptr->length = tempp2->length;
1334                                 d = (mib2_tcp_t *)calloc(tempp2->length, 1);
1335                                 if (d == NULL)
1336                                         goto mibdiff_out_of_memory;
1337                                 diffptr->valp = d;
1338                                 d->tcpRtoMin = t2->tcpRtoMin;
1339                                 d->tcpRtoMax = t2->tcpRtoMax;
1340                                 d->tcpMaxConn = t2->tcpMaxConn;
1341                                 MDIFF(d, t2, t1, tcpActiveOpens);
1342                                 MDIFF(d, t2, t1, tcpPassiveOpens);
1343                                 MDIFF(d, t2, t1, tcpAttemptFails);
1344                                 MDIFF(d, t2, t1, tcpEstabResets);
1345                                 d->tcpCurrEstab = t2->tcpCurrEstab;
1346                                 MDIFF(d, t2, t1, tcpHCOutSegs);
1347                                 MDIFF(d, t2, t1, tcpOutDataSegs);
1348                                 MDIFF(d, t2, t1, tcpOutDataBytes);
1349                                 MDIFF(d, t2, t1, tcpRetransSegs);
1350                                 MDIFF(d, t2, t1, tcpRetransBytes);
1351                                 MDIFF(d, t2, t1, tcpOutAck);
1352                                 MDIFF(d, t2, t1, tcpOutAckDelayed);
1353                                 MDIFF(d, t2, t1, tcpOutUrg);
1354                                 MDIFF(d, t2, t1, tcpOutWinUpdate);
1355                                 MDIFF(d, t2, t1, tcpOutWinProbe);
1356                                 MDIFF(d, t2, t1, tcpOutControl);
1357                                 MDIFF(d, t2, t1, tcpOutRsts);
1358                                 MDIFF(d, t2, t1, tcpOutFastRetrans);
1359                                 MDIFF(d, t2, t1, tcpHCInSegs);
1360                                 MDIFF(d, t2, t1, tcpInAckSegs);
1361                                 MDIFF(d, t2, t1, tcpInAckBytes);
1362                                 MDIFF(d, t2, t1, tcpInDupAck);
1363                                 MDIFF(d, t2, t1, tcpInAckUnsent);
1364                                 MDIFF(d, t2, t1, tcpInDataInorderSegs);
1365                                 MDIFF(d, t2, t1, tcpInDataInorderBytes);
1366                                 MDIFF(d, t2, t1, tcpInDataUnorderSegs);
1367                                 MDIFF(d, t2, t1, tcpInDataUnorderBytes);
1368                                 MDIFF(d, t2, t1, tcpInDataDupSegs);
1369                                 MDIFF(d, t2, t1, tcpInDataDupBytes);
1370                                 MDIFF(d, t2, t1, tcpInDataPartDupSegs);
1371                                 MDIFF(d, t2, t1, tcpInDataPartDupBytes);
1372                                 MDIFF(d, t2, t1, tcpInDataPastWinSegs);
1373                                 MDIFF(d, t2, t1, tcpInDataPastWinBytes);
1374                                 MDIFF(d, t2, t1, tcpInWinProbe);
1375                                 MDIFF(d, t2, t1, tcpInWinUpdate);
1376                                 MDIFF(d, t2, t1, tcpInClosed);
1377                                 MDIFF(d, t2, t1, tcpRttNoUpdate);
1378                                 MDIFF(d, t2, t1, tcpRttUpdate);
1379                                 MDIFF(d, t2, t1, tcpTimRetrans);
1380                                 MDIFF(d, t2, t1, tcpTimRetransDrop);
1381                                 MDIFF(d, t2, t1, tcpTimKeepalive);
1382                                 MDIFF(d, t2, t1, tcpTimKeepaliveProbe);
1383                                 MDIFF(d, t2, t1, tcpTimKeepaliveDrop);
```



```
1384                                 MDIFF(d, t2, t1, tcpListenDrop);
1385                                 MDIFF(d, t2, t1, tcpListenDropQ0);
1386                                 MDIFF(d, t2, t1, tcpHalfOpenDrop);
1387                                 MDIFF(d, t2, t1, tcpOutSackRetransSegs);
1388                                 prevp = diffptr++;
1389                                 break;
1390                         }
1391                         case MIB2_UDP: {
1392                                 mib2_udp_t *u2;
1393                                 mib2_udp_t *u1;
1394                                 mib2_udp_t *d;

1396                                 u2 = (mib2_udp_t *)tempp2->valp;
1397                                 u1 = (mib2_udp_t *)tempp1->valp;
1398                                 diffptr->group = tempp2->group;
1399                                 diffptr->mib_id = tempp2->mib_id;
1400                                 diffptr->length = tempp2->length;
1401                                 d = (mib2_udp_t *)calloc(tempp2->length, 1);
1402                                 if (d == NULL)
1403                                         goto mibdiff_out_of_memory;
1404                                 diffptr->valp = d;
1405                                 MDIFF(d, u2, u1, udpHCInDatagrams);
1406                                 MDIFF(d, u2, u1, udpInErrors);
1407                                 MDIFF(d, u2, u1, udpHCOutDatagrams);
1408                                 MDIFF(d, u2, u1, udpOutErrors);
1409                                 prevp = diffptr++;
1410                                 break;
1411                         }
1412                         case MIB2_SCTP: {
1413                                 mib2_sctp_t *s2;
1414                                 mib2_sctp_t *s1;
1415                                 mib2_sctp_t *d;

1417                                 s2 = (mib2_sctp_t *)tempp2->valp;
1418                                 s1 = (mib2_sctp_t *)tempp1->valp;
1419                                 diffptr->group = tempp2->group;
1420                                 diffptr->mib_id = tempp2->mib_id;
1421                                 diffptr->length = tempp2->length;
1422                                 d = (mib2_sctp_t *)calloc(tempp2->length, 1);
1423                                 if (d == NULL)
1424                                         goto mibdiff_out_of_memory;
1425                                 diffptr->valp = d;
1426                                 d->sctpRtoAlgorithm = s2->sctpRtoAlgorithm;
1427                                 d->sctpRtoMin = s2->sctpRtoMin;
1428                                 d->sctpRtoMax = s2->sctpRtoMax;
1429                                 d->sctpRtoInitial = s2->sctpRtoInitial;
1430                                 d->sctpMaxAssocs = s2->sctpMaxAssocs;
1431                                 d->sctpValCookieLife = s2->sctpValCookieLife;
1432                                 d->sctpMaxInitRetr = s2->sctpMaxInitRetr;
1433                                 d->sctpCurrEstab = s2->sctpCurrEstab;
1434                                 MDIFF(d, s2, s1, sctpActiveEstab);
1435                                 MDIFF(d, s2, s1, sctpPassiveEstab);
1436                                 MDIFF(d, s2, s1, sctpAborted);
1437                                 MDIFF(d, s2, s1, sctpShutdowns);
1438                                 MDIFF(d, s2, s1, sctpOutOfBlue);
1439                                 MDIFF(d, s2, s1, sctpChecksumError);
1440                                 MDIFF(d, s2, s1, sctpOutCtrlChunks);
1441                                 MDIFF(d, s2, s1, sctpOutOrderChunks);
1442                                 MDIFF(d, s2, s1, sctpOutUnorderChunks);
1443                                 MDIFF(d, s2, s1, sctpRetransChunks);
1444                                 MDIFF(d, s2, s1, sctpOutAck);
1445                                 MDIFF(d, s2, s1, sctpOutAckDelayed);
1446                                 MDIFF(d, s2, s1, sctpOutWinUpdate);
1447                                 MDIFF(d, s2, s1, sctpOutFastRetrans);
1448                                 MDIFF(d, s2, s1, sctpOutWinProbe);
1449                                 MDIFF(d, s2, s1, sctpInCtrlChunks);
```

```
1450                                MDIFF(d, s2, s1, sctpInOrderChunks);
1451                                MDIFF(d, s2, s1, sctpInUnorderChunks);
1452                                MDIFF(d, s2, s1, sctpInAck);
1453                                MDIFF(d, s2, s1, sctpInDupAck);
1454                                MDIFF(d, s2, s1, sctpInAckUnsent);
1455                                MDIFF(d, s2, s1, sctpFragUsrMsgs);
1456                                MDIFF(d, s2, s1, sctpReasmUsrMsgs);
1457                                MDIFF(d, s2, s1, sctpOutSCTPPkts);
1458                                MDIFF(d, s2, s1, sctpInSCTPPkts);
1459                                MDIFF(d, s2, s1, sctpInInvalidCookie);
1460                                MDIFF(d, s2, s1, sctpTimRetrans);
1461                                MDIFF(d, s2, s1, sctpTimRetransDrop);
1462                                MDIFF(d, s2, s1, sctpTimHeartBeatProbe);
1463                                MDIFF(d, s2, s1, sctpTimHeartBeatDrop);
1464                                MDIFF(d, s2, s1, sctpListenDrop);
1465                                MDIFF(d, s2, s1, sctpInClosed);
1466                                prevp = diffptr++;
1467                                break;
1468                        }
1469                        case MIB2_DCCP: {
1470                                /* XXX:DCCP */
1471                                break;
1472                        }
1473 #endif /* ! codereview */
1474                        case EXPER_RAWIP: {
1475                                mib2_rawip_t *r2;
1476                                mib2_rawip_t *r1;
1477                                mib2_rawip_t *d;

1479                                r2 = (mib2_rawip_t *)tempp2->valp;
1480                                r1 = (mib2_rawip_t *)tempp1->valp;
1481                                diffptr->group = tempp2->group;
1482                                diffptr->mib_id = tempp2->mib_id;
1483                                diffptr->length = tempp2->length;
1484                                d = (mib2_rawip_t *)calloc(tempp2->length, 1);
1485                                if (d == NULL)
1486                                        goto mibdiff_out_of_memory;
1487                                diffptr->valp = d;
1488                                MDIFF(d, r2, r1, rawipInDatagrams);
1489                                MDIFF(d, r2, r1, rawipInErrors);
1490                                MDIFF(d, r2, r1, rawipInCksumErrs);
1491                                MDIFF(d, r2, r1, rawipOutDatagrams);
1492                                MDIFF(d, r2, r1, rawipOutErrors);
1493                                prevp = diffptr++;
1494                                break;
1495                        }
1496                        /*
1497                         * there are more "group" types but they aren't
1498                         * required for the -s and -Ms options
1499                         */
1500                        }
1501                } /* 'for' loop 2 ends */
1502                tempp1 = NULL;
1503        } /* 'for' loop 1 ends */
1504        tempp2 = NULL;
1505        diffptr--;
1506        diffptr->next_item = NULL;
1507        return (diffp);

1509 mibdiff_out_of_memory:;
1510        mib_item_destroy(&diffp);
1511        return (NULL);
1512 }

1514 /*
1515  * mib_item_destroy: cleans up a mib_item_t *
```



```
1516  * that was created by calling mib_item_dup or
1517  * mib_item_diff
1518  */
1519 static void
1520 mib_item_destroy(mib_item_t **itemp) {
1521        int     nitems = 0;
1522        int     c = 0;
1523        mib_item_t *tempp;

1525        if (itemp == NULL || *itemp == NULL)
1526                return;

1528        for (tempp = *itemp; tempp != NULL; tempp = tempp->next_item)
1529                if (tempp->mib_id == 0)
1530                        nitems++;
1531                else
1532                        return; /* cannot destroy! */

1534        if (nitems == 0)
1535                return;         /* cannot destroy! */

1537        for (c = nitems - 1; c >= 0; c--) {
1538                if ((itemp[0][c]).valp != NULL)
1539                        free((itemp[0][c]).valp);
1540        }
1541        free(*itemp);

1543        *itemp = NULL;
1544 }

1546 /* Compare two Octet_ts.  Return B_TRUE if they match, B_FALSE if not. */
1547 static boolean_t
1548 octetstrmatch(const Octet_t *a, const Octet_t *b)
1549 {
1550        if (a == NULL || b == NULL)
1551                return (B_FALSE);

1553        if (a->o_length != b->o_length)
1554                return (B_FALSE);

1556        return (memcmp(a->o_bytes, b->o_bytes, a->o_length) == 0);
1557 }

1559 /* If octetstr() changes make an appropriate change to STR_EXPAND */
1560 static char *
1561 octetstr(const Octet_t *op, int code, char *dst, uint_t dstlen)
1562 {
1563        int     i;
1564        char    *cp;

1566        cp = dst;
1567        if (op) {
1568                for (i = 0; i < op->o_length; i++) {
1569                        switch (code) {
1570                        case 'd':
1571                                if (cp - dst + 4 > dstlen) {
1572                                        *cp = '\0';
1573                                        return (dst);
1574                                }
1575                                (void) snprintf(cp, 5, "%d.",
1576                                    0xff & op->o_bytes[i]);
1577                                cp = strchr(cp, '\0');
1578                                break;
1579                        case 'a':
1580                                if (cp - dst + 1 > dstlen) {
1581                                        *cp = '\0';
```

```
1582                                        return (dst);
1583                                }
1584                                *cp++ = op->o_bytes[i];
1585                                break;
1586                        case 'h':
1587                        default:
1588                                if (cp - dst + 3 > dstlen) {
1589                                        *cp = '\0';
1590                                        return (dst);
1591                                }
1592                                (void) snprintf(cp, 4, "%02x:",
1593                                    0xff & op->o_bytes[i]);
1594                                cp += 3;
1595                                break;
1596                        }
1597                }
1598        }
1599        if (code != 'a' && cp != dst)
1600                cp--;
1601        *cp = '\0';
1602        return (dst);
1603 }

1605 static const char *
1606 mitcp_state(int state, const mib2_transportMLPEntry_t *attr)
1607 {
1608        static char tcpsbuf[50];
1609        const char *cp;

1611        switch (state) {
1612        case TCPS_CLOSED:
1613                cp = "CLOSED";
1614                break;
1615        case TCPS_IDLE:
1616                cp = "IDLE";
1617                break;
1618        case TCPS_BOUND:
1619                cp = "BOUND";
1620                break;
1621        case TCPS_LISTEN:
1622                cp = "LISTEN";
1623                break;
1624        case TCPS_SYN_SENT:
1625                cp = "SYN_SENT";
1626                break;
1627        case TCPS_SYN_RCVD:
1628                cp = "SYN_RCVD";
1629                break;
1630        case TCPS_ESTABLISHED:
1631                cp = "ESTABLISHED";
1632                break;
1633        case TCPS_CLOSE_WAIT:
1634                cp = "CLOSE_WAIT";
1635                break;
1636        case TCPS_FIN_WAIT_1:
1637                cp = "FIN_WAIT_1";
1638                break;
1639        case TCPS_CLOSING:
1640                cp = "CLOSING";
1641                break;
1642        case TCPS_LAST_ACK:
1643                cp = "LAST_ACK";
1644                break;
1645        case TCPS_FIN_WAIT_2:
1646                cp = "FIN_WAIT_2";
1647                break;
```



```
1648        case TCPS_TIME_WAIT:
1649                cp = "TIME_WAIT";
1650                break;
1651        default:
1652                (void) snprintf(tcpsbuf, sizeof (tcpsbuf),
1653                    "UnknownState(%d)", state);
1654                cp = tcpsbuf;
1655                break;
1656        }

1658        if (RSECflag && attr != NULL && attr->tme_flags != 0) {
1659                if (cp != tcpsbuf) {
1660                        (void) strlcpy(tcpsbuf, cp, sizeof (tcpsbuf));
1661                        cp = tcpsbuf;
1662                }
1663                if (attr->tme_flags & MIB2_TMEF_PRIVATE)
1664                        (void) strlcat(tcpsbuf, " P", sizeof (tcpsbuf));
1665                if (attr->tme_flags & MIB2_TMEF_SHARED)
1666                        (void) strlcat(tcpsbuf, " S", sizeof (tcpsbuf));
1667        }

1669        return (cp);
1670 }

1672 static const char *
1673 miudp_state(int state, const mib2_transportMLPEntry_t *attr)
1674 {
1675        static char udpsbuf[50];
1676        const char *cp;

1678        switch (state) {
1679        case MIB2_UDP_unbound:
1680                cp = "Unbound";
1681                break;
1682        case MIB2_UDP_idle:
1683                cp = "Idle";
1684                break;
1685        case MIB2_UDP_connected:
1686                cp = "Connected";
1687                break;
1688        default:
1689                (void) snprintf(udpsbuf, sizeof (udpsbuf),
1690                    "Unknown State(%d)", state);
1691                cp = udpsbuf;
1692                break;
1693        }

1695        if (RSECflag && attr != NULL && attr->tme_flags != 0) {
1696                if (cp != udpsbuf) {
1697                        (void) strlcpy(udpsbuf, cp, sizeof (udpsbuf));
1698                        cp = udpsbuf;
1699                }
1700                if (attr->tme_flags & MIB2_TMEF_PRIVATE)
1701                        (void) strlcat(udpsbuf, " P", sizeof (udpsbuf));
1702                if (attr->tme_flags & MIB2_TMEF_SHARED)
1703                        (void) strlcat(udpsbuf, " S", sizeof (udpsbuf));
1704        }

1706        return (cp);
1707 }

1709 static int odd;

1711 static void
1712 prval_init(void)
1713 {
```

```
1714         odd = 0;
1715 }

1717 static void
1718 prval(char *str, Counter val)
1719 {
1720         (void) printf("\t%-20s=%6u", str, val);
1721         if (odd++ & 1)
1722                 (void) putchar('\n');
1723 }

1725 static void
1726 prval64(char *str, Counter64 val)
1727 {
1728         (void) printf("\t%-20s=%6llu", str, val);
1729         if (odd++ & 1)
1730                 (void) putchar('\n');
1731 }

1733 static void
1734 pr_int_val(char *str, int val)
1735 {
1736         (void) printf("\t%-20s=%6d", str, val);
1737         if (odd++ & 1)
1738                 (void) putchar('\n');
1739 }

1741 static void
1742 pr_sctp_rtoalgo(char *str, int val)
1743 {
1744         (void) printf("\t%-20s=", str);
1745         switch (val) {
1746                 case MIB2_SCTP_RTOALGO_OTHER:
1747                         (void) printf("%6.6s", "other");
1748                         break;

1750                 case MIB2_SCTP_RTOALGO_VANJ:
1751                         (void) printf("%6.6s", "vanj");
1752                         break;

1754                 default:
1755                         (void) printf("%6d", val);
1756                         break;
1757         }
1758         if (odd++ & 1)
1759                 (void) putchar('\n');
1760 }

1762 static void
1763 prval_end(void)
1764 {
1765         if (odd++ & 1)
1766                 (void) putchar('\n');
1767 }

1769 /* Extract constant sizes */
1770 static void
1771 mib_get_constants(mib_item_t *item)
1772 {
1773         /* 'for' loop 1: */
1774         for (; item; item = item->next_item) {
1775                 if (item->mib_id != 0)
1776                         continue; /* 'for' loop 1 */

1778                 switch (item->group) {
1779                 case MIB2_IP: {
```



```
1780                         mib2_ip_t       *ip = (mib2_ip_t *)item->valp;

1782                         ipAddrEntrySize = ip->ipAddrEntrySize;
1783                         ipRouteEntrySize = ip->ipRouteEntrySize;
1784                         ipNetToMediaEntrySize = ip->ipNetToMediaEntrySize;
1785                         ipMemberEntrySize = ip->ipMemberEntrySize;
1786                         ipGroupSourceEntrySize = ip->ipGroupSourceEntrySize;
1787                         ipRouteAttributeSize = ip->ipRouteAttributeSize;
1788                         transportMLPSize = ip->transportMLPSize;
1789                         ipDestEntrySize = ip->ipDestEntrySize;
1790                         assert(IS_P2ALIGNED(ipAddrEntrySize,
1791                             sizeof (mib2_ipAddrEntry_t *)));
1792                         assert(IS_P2ALIGNED(ipRouteEntrySize,
1793                             sizeof (mib2_ipRouteEntry_t *)));
1794                         assert(IS_P2ALIGNED(ipNetToMediaEntrySize,
1795                             sizeof (mib2_ipNetToMediaEntry_t *)));
1796                         assert(IS_P2ALIGNED(ipMemberEntrySize,
1797                             sizeof (ip_member_t *)));
1798                         assert(IS_P2ALIGNED(ipGroupSourceEntrySize,
1799                             sizeof (ip_grpsrc_t *)));
1800                         assert(IS_P2ALIGNED(ipRouteAttributeSize,
1801                             sizeof (mib2_ipAttributeEntry_t *)));
1802                         assert(IS_P2ALIGNED(transportMLPSize,
1803                             sizeof (mib2_transportMLPEntry_t *)));
1804                         break;
1805                 }
1806                 case EXPER_DVMRP: {
1807                         struct mrtstat  *mrts = (struct mrtstat *)item->valp;

1809                         vifctlSize = mrts->mrts_vifctlSize;
1810                         mfcctlSize = mrts->mrts_mfcctlSize;
1811                         assert(IS_P2ALIGNED(vifctlSize,
1812                             sizeof (struct vifclt *)));
1813                         assert(IS_P2ALIGNED(mfcctlSize,
1814                             sizeof (struct mfcctl *)));
1815                         break;
1816                 }
1817                 case MIB2_IP6: {
1818                         mib2_ipv6IfStatsEntry_t *ip6;
1819                         /* Just use the first entry */

1821                         ip6 = (mib2_ipv6IfStatsEntry_t *)item->valp;
1822                         ipv6IfStatsEntrySize = ip6->ipv6IfStatsEntrySize;
1823                         ipv6AddrEntrySize = ip6->ipv6AddrEntrySize;
1824                         ipv6RouteEntrySize = ip6->ipv6RouteEntrySize;
1825                         ipv6NetToMediaEntrySize = ip6->ipv6NetToMediaEntrySize;
1826                         ipv6MemberEntrySize = ip6->ipv6MemberEntrySize;
1827                         ipv6GroupSourceEntrySize =
1828                             ip6->ipv6GroupSourceEntrySize;
1829                         assert(IS_P2ALIGNED(ipv6IfStatsEntrySize,
1830                             sizeof (mib2_ipv6IfStatsEntry_t *)));
1831                         assert(IS_P2ALIGNED(ipv6AddrEntrySize,
1832                             sizeof (mib2_ipv6AddrEntry_t *)));
1833                         assert(IS_P2ALIGNED(ipv6RouteEntrySize,
1834                             sizeof (mib2_ipv6RouteEntry_t *)));
1835                         assert(IS_P2ALIGNED(ipv6NetToMediaEntrySize,
1836                             sizeof (mib2_ipv6NetToMediaEntry_t *)));
1837                         assert(IS_P2ALIGNED(ipv6MemberEntrySize,
1838                             sizeof (ipv6_member_t *)));
1839                         assert(IS_P2ALIGNED(ipv6GroupSourceEntrySize,
1840                             sizeof (ipv6_grpsrc_t *)));
1841                         break;
1842                 }
1843                 case MIB2_ICMP6: {
1844                         mib2_ipv6IfIcmpEntry_t *icmp6;
1845                         /* Just use the first entry */
```

```
1847                        icmp6 = (mib2_ipv6IfIcmpEntry_t *)item->valp;
1848                        ipv6IfIcmpEntrySize = icmp6->ipv6IfIcmpEntrySize;
1849                        assert(IS_P2ALIGNED(ipv6IfIcmpEntrySize,
1850                            sizeof (mib2_ipv6IfIcmpEntry_t *)));
1851                        break;
1852                }
1853                case MIB2_TCP: {
1854                        mib2_tcp_t      *tcp = (mib2_tcp_t *)item->valp;

1856                        tcpConnEntrySize = tcp->tcpConnTableSize;
1857                        tcp6ConnEntrySize = tcp->tcp6ConnTableSize;
1858                        assert(IS_P2ALIGNED(tcpConnEntrySize,
1859                            sizeof (mib2_tcpConnEntry_t *)));
1860                        assert(IS_P2ALIGNED(tcp6ConnEntrySize,
1861                            sizeof (mib2_tcp6ConnEntry_t *)));
1862                        break;
1863                }
1864                case MIB2_UDP: {
1865                        mib2_udp_t      *udp = (mib2_udp_t *)item->valp;

1867                        udpEntrySize = udp->udpEntrySize;
1868                        udp6EntrySize = udp->udp6EntrySize;
1869                        assert(IS_P2ALIGNED(udpEntrySize,
1870                            sizeof (mib2_udpEntry_t *)));
1871                        assert(IS_P2ALIGNED(udp6EntrySize,
1872                            sizeof (mib2_udp6Entry_t *)));
1873                        break;
1874                }
1875                case MIB2_SCTP: {
1876                        mib2_sctp_t     *sctp = (mib2_sctp_t *)item->valp;

1878                        sctpEntrySize = sctp->sctpEntrySize;
1879                        sctpLocalEntrySize = sctp->sctpLocalEntrySize;
1880                        sctpRemoteEntrySize = sctp->sctpRemoteEntrySize;
1881                        break;
1882                }
1883                case MIB2_DCCP: {
1884                        mib2_dccp_t     *dccp = (mib2_dccp_t *)item->valp;

1886                        dccpEntrySize = dccp->dccpConnTableSize;
1887                        dccp6EntrySize = dccp->dccp6ConnTableSize;
1888                        assert(IS_P2ALIGNED(dccpEntrySize,
1889                            sizeof (mib2_dccpConnEntry_t *)));
1890                        assert(IS_P2ALIGNED(dccp6EntrySize,
1891                            sizeof (mib2_dccp6ConnEntry_t *)));
1892                        break;
1893                }
1894 #endif /* ! codereview */
1895                }
1896        } /* 'for' loop 1 ends */

1898        if (Xflag) {
1899                (void) puts("mib_get_constants:");
1900                (void) printf("\tipv6IfStatsEntrySize %d\n",
1901                    ipv6IfStatsEntrySize);
1902                (void) printf("\tipAddrEntrySize %d\n", ipAddrEntrySize);
1903                (void) printf("\tipRouteEntrySize %d\n", ipRouteEntrySize);
1904                (void) printf("\tipNetToMediaEntrySize %d\n",
1905                    ipNetToMediaEntrySize);
1906                (void) printf("\tipMemberEntrySize %d\n", ipMemberEntrySize);
1907                (void) printf("\tipRouteAttributeSize %d\n",
1908                    ipRouteAttributeSize);
1909                (void) printf("\tvifctlSize %d\n", vifctlSize);
1910                (void) printf("\tmfcctlSize %d\n", mfcctlSize);
```



```
1912                (void) printf("\tipv6AddrEntrySize %d\n", ipv6AddrEntrySize);
1913                (void) printf("\tipv6RouteEntrySize %d\n", ipv6RouteEntrySize);
1914                (void) printf("\tipv6NetToMediaEntrySize %d\n",
1915                    ipv6NetToMediaEntrySize);
1916                (void) printf("\tipv6MemberEntrySize %d\n",
1917                    ipv6MemberEntrySize);
1918                (void) printf("\tipv6IfIcmpEntrySize %d\n",
1919                    ipv6IfIcmpEntrySize);
1920                (void) printf("\tipDestEntrySize %d\n", ipDestEntrySize);
1921                (void) printf("\ttransportMLPSize %d\n", transportMLPSize);
1922                (void) printf("\ttcpConnEntrySize %d\n", tcpConnEntrySize);
1923                (void) printf("\ttcp6ConnEntrySize %d\n", tcp6ConnEntrySize);
1924                (void) printf("\tudpEntrySize %d\n", udpEntrySize);
1925                (void) printf("\tudp6EntrySize %d\n", udp6EntrySize);
1926                (void) printf("\tsctpEntrySize %d\n", sctpEntrySize);
1927                (void) printf("\tsctpLocalEntrySize %d\n", sctpLocalEntrySize);
1928                (void) printf("\tsctpRemoteEntrySize %d\n",
1929                    sctpRemoteEntrySize);
1930                (void) printf("\tdccpEntrySize %d\n", dccpEntrySize);
1931                (void) printf("\ttcp6EntrySize %d\n", dccp6EntrySize);
1932 #endif /* ! codereview */
1933        }
1934 }


1937 /* ---------------------------- STAT_REPORT ------------------------------ */

1939 static void
1940 stat_report(mib_item_t *item)
1941 {
1942        int     jtemp = 0;
1943        char    ifname[LIFNAMSIZ + 1];

1945        /* 'for' loop 1: */
1946        for (; item; item = item->next_item) {
1947                if (Xflag) {
1948                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
1949                        (void) printf("Group = %d, mib_id = %d, "
1950                            "length = %d, valp = 0x%p\n",
1951                            item->group, item->mib_id,
1952                            item->length, item->valp);
1953                }
1954                if (item->mib_id != 0)
1955                        continue; /* 'for' loop 1 */

1957                switch (item->group) {
1958                case MIB2_IP: {
1959                        mib2_ip_t       *ip = (mib2_ip_t *)item->valp;

1961                        if (protocol_selected(IPPROTO_IP) &&
1962                            family_selected(AF_INET)) {
1963                                (void) fputs(v4compat ? "\nIP" : "\nIPv4",
1964                                    stdout);
1965                                print_ip_stats(ip);
1966                        }
1967                        break;
1968                }
1969                case MIB2_ICMP: {
1970                        mib2_icmp_t     *icmp =
1971                            (mib2_icmp_t *)item->valp;

1973                        if (protocol_selected(IPPROTO_ICMP) &&
1974                            family_selected(AF_INET)) {
1975                                (void) fputs(v4compat ? "\nICMP" : "\nICMPv4",
1976                                    stdout);
1977                                print_icmp_stats(icmp);
```

```
1978                        }
1979                        break;
1980                }
1981                case MIB2_IP6: {
1982                        mib2_ipv6IfStatsEntry_t *ip6;
1983                        mib2_ipv6IfStatsEntry_t sum6;

1985                        if (!(protocol_selected(IPPROTO_IPV6)) ||
1986                            !(family_selected(AF_INET6)))
1987                                break;
1988                        bzero(&sum6, sizeof (sum6));
1989                        /* 'for' loop 2a: */
1990                        for (ip6 = (mib2_ipv6IfStatsEntry_t *)item->valp;
1991                            (char *)ip6 < (char *)item->valp + item->length;
1992                            /* LINTED: (note 1) */
1993                            ip6 = (mib2_ipv6IfStatsEntry_t *)((char *)ip6 +
1994                            ipv6IfStatsEntrySize)) {
1995                                if (ip6->ipv6IfIndex == 0) {
1996                                        /*
1997                                         * The "unknown interface" ip6
1998                                         * mib. Just add to the sum.
1999                                         */
2000                                        sum_ip6_stats(ip6, &sum6);
2001                                        continue; /* 'for' loop 2a */
2002                                }
2003                                if (Aflag) {
2004                                        (void) printf("\nIPv6 for %s\n",
2005                                            ifindex2str(ip6->ipv6IfIndex,
2006                                            ifname));
2007                                        print_ip6_stats(ip6);
2008                                }
2009                                sum_ip6_stats(ip6, &sum6);
2010                        } /* 'for' loop 2a ends */
2011                        (void) fputs("\nIPv6", stdout);
2012                        print_ip6_stats(&sum6);
2013                        break;
2014                }
2015                case MIB2_ICMP6: {
2016                        mib2_ipv6IfIcmpEntry_t *icmp6;
2017                        mib2_ipv6IfIcmpEntry_t sum6;

2019                        if (!(protocol_selected(IPPROTO_ICMPV6)) ||
2020                            !(family_selected(AF_INET6)))
2021                                break;
2022                        bzero(&sum6, sizeof (sum6));
2023                        /* 'for' loop 2b: */
2024                        for (icmp6 = (mib2_ipv6IfIcmpEntry_t *)item->valp;
2025                            (char *)icmp6 < (char *)item->valp + item->length;
2026                            icmp6 = (void *)((char *)icmp6 +
2027                            ipv6IfIcmpEntrySize)) {
2028                                if (icmp6->ipv6IfIcmpIfIndex == 0) {
2029                                        /*
2030                                         * The "unknown interface" icmp6
2031                                         * mib. Just add to the sum.
2032                                         */
2033                                        sum_icmp6_stats(icmp6, &sum6);
2034                                        continue; /* 'for' loop 2b: */
2035                                }
2036                                if (Aflag) {
2037                                        (void) printf("\nICMPv6 for %s\n",
2038                                            ifindex2str(
2039                                            icmp6->ipv6IfIcmpIfIndex, ifname));
2040                                        print_icmp6_stats(icmp6);
2041                                }
2042                                sum_icmp6_stats(icmp6, &sum6);
2043                        } /* 'for' loop 2b ends */
```



```
2044                        (void) fputs("\nICMPv6", stdout);
2045                        print_icmp6_stats(&sum6);
2046                        break;
2047                }
2048                case MIB2_TCP: {
2049                        mib2_tcp_t      *tcp = (mib2_tcp_t *)item->valp;

2051                        if (protocol_selected(IPPROTO_TCP) &&
2052                            (family_selected(AF_INET) ||
2053                            family_selected(AF_INET6))) {
2054                                (void) fputs("\nTCP", stdout);
2055                                print_tcp_stats(tcp);
2056                        }
2057                        break;
2058                }
2059                case MIB2_UDP: {
2060                        mib2_udp_t      *udp = (mib2_udp_t *)item->valp;

2062                        if (protocol_selected(IPPROTO_UDP) &&
2063                            (family_selected(AF_INET) ||
2064                            family_selected(AF_INET6))) {
2065                                (void) fputs("\nUDP", stdout);
2066                                print_udp_stats(udp);
2067                        }
2068                        break;
2069                }
2070                case MIB2_SCTP: {
2071                        mib2_sctp_t     *sctp = (mib2_sctp_t *)item->valp;

2073                        if (protocol_selected(IPPROTO_SCTP) &&
2074                            (family_selected(AF_INET) ||
2075                            family_selected(AF_INET6))) {
2076                                (void) fputs("\nSCTP", stdout);
2077                                print_sctp_stats(sctp);
2078                        }
2079                        break;
2080                }
2081                case EXPER_RAWIP: {
2082                        mib2_rawip_t    *rawip =
2083                            (mib2_rawip_t *)item->valp;

2085                        if (protocol_selected(IPPROTO_RAW) &&
2086                            (family_selected(AF_INET) ||
2087                            family_selected(AF_INET6))) {
2088                                (void) fputs("\nRAWIP", stdout);
2089                                print_rawip_stats(rawip);
2090                        }
2091                        break;
2092                }
2093                case EXPER_IGMP: {
2094                        struct igmpstat *igps =
2095                            (struct igmpstat *)item->valp;

2097                        if (protocol_selected(IPPROTO_IGMP) &&
2098                            (family_selected(AF_INET))) {
2099                                (void) fputs("\nIGMP:\n", stdout);
2100                                print_igmp_stats(igps);
2101                        }
2102                        break;
2103                }
2104                }
2105        } /* 'for' loop 1 ends */
2106        (void) putchar('\n');
2107        (void) fflush(stdout);
2108 }
```

```
2110 static void
2111 print_ip_stats(mib2_ip_t *ip)
2112 {
2113         prval_init();
2114         pr_int_val("ipForwarding",      ip->ipForwarding);
2115         pr_int_val("ipDefaultTTL",      ip->ipDefaultTTL);
2116         prval("ipInReceives",           ip->ipInReceives);
2117         prval("ipInHdrErrors",          ip->ipInHdrErrors);
2118         prval("ipInAddrErrors",         ip->ipInAddrErrors);
2119         prval("ipInCksumErrs",          ip->ipInCksumErrs);
2120         prval("ipForwDatagrams",        ip->ipForwDatagrams);
2121         prval("ipForwProhibits",        ip->ipForwProhibits);
2122         prval("ipInUnknownProtos",      ip->ipInUnknownProtos);
2123         prval("ipInDiscards",           ip->ipInDiscards);
2124         prval("ipInDelivers",           ip->ipInDelivers);
2125         prval("ipOutRequests",          ip->ipOutRequests);
2126         prval("ipOutDiscards",          ip->ipOutDiscards);
2127         prval("ipOutNoRoutes",          ip->ipOutNoRoutes);
2128         pr_int_val("ipReasmTimeout",    ip->ipReasmTimeout);
2129         prval("ipReasmReqds",           ip->ipReasmReqds);
2130         prval("ipReasmOKs",             ip->ipReasmOKs);
2131         prval("ipReasmFails",           ip->ipReasmFails);
2132         prval("ipReasmDuplicates",      ip->ipReasmDuplicates);
2133         prval("ipReasmPartDups",        ip->ipReasmPartDups);
2134         prval("ipFragOKs",              ip->ipFragOKs);
2135         prval("ipFragFails",            ip->ipFragFails);
2136         prval("ipFragCreates",          ip->ipFragCreates);
2137         prval("ipRoutingDiscards",      ip->ipRoutingDiscards);

2139         prval("tcpInErrs",              ip->tcpInErrs);
2140         prval("udpNoPorts",             ip->udpNoPorts);
2141         prval("udpInCksumErrs",         ip->udpInCksumErrs);
2142         prval("udpInOverflows",         ip->udpInOverflows);
2143         prval("rawipInOverflows",       ip->rawipInOverflows);
2144         prval("ipsecInSucceeded",       ip->ipsecInSucceeded);
2145         prval("ipsecInFailed",          ip->ipsecInFailed);
2146         prval("ipInIPv6",               ip->ipInIPv6);
2147         prval("ipOutIPv6",              ip->ipOutIPv6);
2148         prval("ipOutSwitchIPv6",        ip->ipOutSwitchIPv6);
2149         prval_end();
2150 }

2152 static void
2153 print_icmp_stats(mib2_icmp_t *icmp)
2154 {
2155         prval_init();
2156         prval("icmpInMsgs",             icmp->icmpInMsgs);
2157         prval("icmpInErrors",           icmp->icmpInErrors);
2158         prval("icmpInCksumErrs",        icmp->icmpInCksumErrs);
2159         prval("icmpInUnknowns",         icmp->icmpInUnknowns);
2160         prval("icmpInDestUnreachs",     icmp->icmpInDestUnreachs);
2161         prval("icmpInTimeExcds",        icmp->icmpInTimeExcds);
2162         prval("icmpInParmProbs",        icmp->icmpInParmProbs);
2163         prval("icmpInSrcQuenchs",       icmp->icmpInSrcQuenchs);
2164         prval("icmpInRedirects",        icmp->icmpInRedirects);
2165         prval("icmpInBadRedirects",     icmp->icmpInBadRedirects);
2166         prval("icmpInEchos",            icmp->icmpInEchos);
2167         prval("icmpInEchoReps",         icmp->icmpInEchoReps);
2168         prval("icmpInTimestamps",       icmp->icmpInTimestamps);
2169         prval("icmpInTimestampReps",    icmp->icmpInTimestampReps);
2170         prval("icmpInAddrMasks",        icmp->icmpInAddrMasks);
2171         prval("icmpInAddrMaskReps",     icmp->icmpInAddrMaskReps);
2172         prval("icmpInFragNeeded",       icmp->icmpInFragNeeded);
2173         prval("icmpOutMsgs",            icmp->icmpOutMsgs);
2174         prval("icmpOutDrops",           icmp->icmpOutDrops);
2175         prval("icmpOutErrors",          icmp->icmpOutErrors);
```



```
2176         prval("icmpOutDestUnreachs",    icmp->icmpOutDestUnreachs);
2177         prval("icmpOutTimeExcds",       icmp->icmpOutTimeExcds);
2178         prval("icmpOutParmProbs",       icmp->icmpOutParmProbs);
2179         prval("icmpOutSrcQuenchs",      icmp->icmpOutSrcQuenchs);
2180         prval("icmpOutRedirects",       icmp->icmpOutRedirects);
2181         prval("icmpOutEchos",           icmp->icmpOutEchos);
2182         prval("icmpOutEchoReps",        icmp->icmpOutEchoReps);
2183         prval("icmpOutTimestamps",      icmp->icmpOutTimestamps);
2184         prval("icmpOutTimestampReps",   icmp->icmpOutTimestampReps);
2185         prval("icmpOutAddrMasks",       icmp->icmpOutAddrMasks);
2186         prval("icmpOutAddrMaskReps",    icmp->icmpOutAddrMaskReps);
2187         prval("icmpOutFragNeeded",      icmp->icmpOutFragNeeded);
2188         prval("icmpInOverflows",        icmp->icmpInOverflows);
2189         prval_end();
2190 }

2192 static void
2193 print_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6)
2194 {
2195         prval_init();
2196         prval("ipv6Forwarding",         ip6->ipv6Forwarding);
2197         prval("ipv6DefaultHopLimit",    ip6->ipv6DefaultHopLimit);

2199         prval("ipv6InReceives",         ip6->ipv6InReceives);
2200         prval("ipv6InHdrErrors",        ip6->ipv6InHdrErrors);
2201         prval("ipv6InTooBigErrors",     ip6->ipv6InTooBigErrors);
2202         prval("ipv6InNoRoutes",         ip6->ipv6InNoRoutes);
2203         prval("ipv6InAddrErrors",       ip6->ipv6InAddrErrors);
2204         prval("ipv6InUnknownProtos",    ip6->ipv6InUnknownProtos);
2205         prval("ipv6InTruncatedPkts",    ip6->ipv6InTruncatedPkts);
2206         prval("ipv6InDiscards",         ip6->ipv6InDiscards);
2207         prval("ipv6InDelivers",         ip6->ipv6InDelivers);
2208         prval("ipv6OutForwDatagrams",   ip6->ipv6OutForwDatagrams);
2209         prval("ipv6OutRequests",        ip6->ipv6OutRequests);
2210         prval("ipv6OutDiscards",        ip6->ipv6OutDiscards);
2211         prval("ipv6OutNoRoutes",        ip6->ipv6OutNoRoutes);
2212         prval("ipv6OutFragOKs",         ip6->ipv6OutFragOKs);
2213         prval("ipv6OutFragFails",       ip6->ipv6OutFragFails);
2214         prval("ipv6OutFragCreates",     ip6->ipv6OutFragCreates);
2215         prval("ipv6ReasmReqds",         ip6->ipv6ReasmReqds);
2216         prval("ipv6ReasmOKs",           ip6->ipv6ReasmOKs);
2217         prval("ipv6ReasmFails",         ip6->ipv6ReasmFails);
2218         prval("ipv6InMcastPkts",        ip6->ipv6InMcastPkts);
2219         prval("ipv6OutMcastPkts",       ip6->ipv6OutMcastPkts);
2220         prval("ipv6ReasmDuplicates",    ip6->ipv6ReasmDuplicates);
2221         prval("ipv6ReasmPartDups",      ip6->ipv6ReasmPartDups);
2222         prval("ipv6ForwProhibits",      ip6->ipv6ForwProhibits);
2223         prval("udpInCksumErrs",         ip6->udpInCksumErrs);
2224         prval("udpInOverflows",         ip6->udpInOverflows);
2225         prval("rawipInOverflows",       ip6->rawipInOverflows);
2226         prval("ipv6InIPv4",             ip6->ipv6InIPv4);
2227         prval("ipv6OutIPv4",            ip6->ipv6OutIPv4);
2228         prval("ipv6OutSwitchIPv4",      ip6->ipv6OutSwitchIPv4);
2229         prval_end();
2230 }

2232 static void
2233 print_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6)
2234 {
2235         prval_init();
2236         prval("icmp6InMsgs",            icmp6->ipv6IfIcmpInMsgs);
2237         prval("icmp6InErrors",          icmp6->ipv6IfIcmpInErrors);
2238         prval("icmp6InDestUnreachs",    icmp6->ipv6IfIcmpInDestUnreachs);
2239         prval("icmp6InAdminProhibs",    icmp6->ipv6IfIcmpInAdminProhibs);
2240         prval("icmp6InTimeExcds",       icmp6->ipv6IfIcmpInTimeExcds);
2241         prval("icmp6InParmProblems",    icmp6->ipv6IfIcmpInParmProblems);
```

```
2242         prval("icmp6InPktTooBigs",      icmp6->ipv6IfIcmpInPktTooBigs);
2243         prval("icmp6InEchos",           icmp6->ipv6IfIcmpInEchos);
2244         prval("icmp6InEchoReplies",     icmp6->ipv6IfIcmpInEchoReplies);
2245         prval("icmp6InRouterSols",      icmp6->ipv6IfIcmpInRouterSolicits);
2246         prval("icmp6InRouterAds",
2247             icmp6->ipv6IfIcmpInRouterAdvertisements);
2248         prval("icmp6InNeighborSols",    icmp6->ipv6IfIcmpInNeighborSolicits);
2249         prval("icmp6InNeighborAds",
2250             icmp6->ipv6IfIcmpInNeighborAdvertisements);
2251         prval("icmp6InRedirects",       icmp6->ipv6IfIcmpInRedirects);
2252         prval("icmp6InBadRedirects",    icmp6->ipv6IfIcmpInBadRedirects);
2253         prval("icmp6InGroupQueries",    icmp6->ipv6IfIcmpInGroupMembQueries);
2254         prval("icmp6InGroupResps",      icmp6->ipv6IfIcmpInGroupMembResponses);
2255         prval("icmp6InGroupReds",       icmp6->ipv6IfIcmpInGroupMembReductions);
2256         prval("icmp6InOverflows",       icmp6->ipv6IfIcmpInOverflows);
2257         prval_end();
2258         prval_init();
2259         prval("icmp6OutMsgs",           icmp6->ipv6IfIcmpOutMsgs);
2260         prval("icmp6OutErrors",         icmp6->ipv6IfIcmpOutErrors);
2261         prval("icmp6OutDestUnreachs",   icmp6->ipv6IfIcmpOutDestUnreachs);
2262         prval("icmp6OutAdminProhibs",   icmp6->ipv6IfIcmpOutAdminProhibs);
2263         prval("icmp6OutTimeExcds",      icmp6->ipv6IfIcmpOutTimeExcds);
2264         prval("icmp6OutParmProblems",   icmp6->ipv6IfIcmpOutParmProblems);
2265         prval("icmp6OutPktTooBigs",     icmp6->ipv6IfIcmpOutPktTooBigs);
2266         prval("icmp6OutEchos",          icmp6->ipv6IfIcmpOutEchos);
2267         prval("icmp6OutEchoReplies",    icmp6->ipv6IfIcmpOutEchoReplies);
2268         prval("icmp6OutRouterSols",     icmp6->ipv6IfIcmpOutRouterSolicits);
2269         prval("icmp6OutRouterAds",
2270             icmp6->ipv6IfIcmpOutRouterAdvertisements);
2271         prval("icmp6OutNeighborSols",   icmp6->ipv6IfIcmpOutNeighborSolicits);
2272         prval("icmp6OutNeighborAds",
2273             icmp6->ipv6IfIcmpOutNeighborAdvertisements);
2274         prval("icmp6OutRedirects",      icmp6->ipv6IfIcmpOutRedirects);
2275         prval("icmp6OutGroupQueries",   icmp6->ipv6IfIcmpOutGroupMembQueries);
2276         prval("icmp6OutGroupResps",
2277             icmp6->ipv6IfIcmpOutGroupMembResponses);
2278         prval("icmp6OutGroupReds",
2279             icmp6->ipv6IfIcmpOutGroupMembReductions);
2280         prval_end();
2281 }

2283 static void
2284 print_sctp_stats(mib2_sctp_t *sctp)
2285 {
2286         prval_init();
2287         pr_sctp_rtoalgo("sctpRtoAlgorithm", sctp->sctpRtoAlgorithm);
2288         prval("sctpRtoMin",             sctp->sctpRtoMin);
2289         prval("sctpRtoMax",             sctp->sctpRtoMax);
2290         prval("sctpRtoInitial",         sctp->sctpRtoInitial);
2291         pr_int_val("sctpMaxAssocs",     sctp->sctpMaxAssocs);
2292         prval("sctpValCookieLife",      sctp->sctpValCookieLife);
2293         prval("sctpMaxInitRetr",        sctp->sctpMaxInitRetr);
2294         prval("sctpCurrEstab",          sctp->sctpCurrEstab);
2295         prval("sctpActiveEstab",        sctp->sctpActiveEstab);
2296         prval("sctpPassiveEstab",       sctp->sctpPassiveEstab);
2297         prval("sctpAborted",            sctp->sctpAborted);
2298         prval("sctpShutdowns",          sctp->sctpShutdowns);
2299         prval("sctpOutOfBlue",          sctp->sctpOutOfBlue);
2300         prval("sctpChecksumError",      sctp->sctpChecksumError);
2301         prval64("sctpOutCtrlChunks",    sctp->sctpOutCtrlChunks);
2302         prval64("sctpOutOrderChunks",   sctp->sctpOutOrderChunks);
2303         prval64("sctpOutUnorderChunks", sctp->sctpOutUnorderChunks);
2304         prval64("sctpRetransChunks",    sctp->sctpRetransChunks);
2305         prval("sctpOutAck",             sctp->sctpOutAck);
2306         prval("sctpOutAckDelayed",      sctp->sctpOutAckDelayed);
2307         prval("sctpOutWinUpdate",       sctp->sctpOutWinUpdate);
```



```
2308         prval("sctpOutFastRetrans",     sctp->sctpOutFastRetrans);
2309         prval("sctpOutWinProbe",        sctp->sctpOutWinProbe);
2310         prval64("sctpInCtrlChunks",     sctp->sctpInCtrlChunks);
2311         prval64("sctpInOrderChunks",    sctp->sctpInOrderChunks);
2312         prval64("sctpInUnorderChunks",  sctp->sctpInUnorderChunks);
2313         prval("sctpInAck",              sctp->sctpInAck);
2314         prval("sctpInDupAck",           sctp->sctpInDupAck);
2315         prval("sctpInAckUnsent",        sctp->sctpInAckUnsent);
2316         prval64("sctpFragUsrMsgs",      sctp->sctpFragUsrMsgs);
2317         prval64("sctpReasmUsrMsgs",     sctp->sctpReasmUsrMsgs);
2318         prval64("sctpOutSCTPPkts",      sctp->sctpOutSCTPPkts);
2319         prval64("sctpInSCTPPkts",       sctp->sctpInSCTPPkts);
2320         prval("sctpInInvalidCookie",    sctp->sctpInInvalidCookie);
2321         prval("sctpTimRetrans",         sctp->sctpTimRetrans);
2322         prval("sctpTimRetransDrop",     sctp->sctpTimRetransDrop);
2323         prval("sctpTimHearBeatProbe",   sctp->sctpTimHeartBeatProbe);
2324         prval("sctpTimHearBeatDrop",    sctp->sctpTimHeartBeatDrop);
2325         prval("sctpListenDrop",         sctp->sctpListenDrop);
2326         prval("sctpInClosed",           sctp->sctpInClosed);
2327         prval_end();
2328 }

2330 static void
2331 print_tcp_stats(mib2_tcp_t *tcp)
2332 {
2333         prval_init();
2334         pr_int_val("tcpRtoAlgorithm",   tcp->tcpRtoAlgorithm);
2335         pr_int_val("tcpRtoMin",         tcp->tcpRtoMin);
2336         pr_int_val("tcpRtoMax",         tcp->tcpRtoMax);
2337         pr_int_val("tcpMaxConn",        tcp->tcpMaxConn);
2338         prval("tcpActiveOpens",         tcp->tcpActiveOpens);
2339         prval("tcpPassiveOpens",        tcp->tcpPassiveOpens);
2340         prval("tcpAttemptFails",        tcp->tcpAttemptFails);
2341         prval("tcpEstabResets",         tcp->tcpEstabResets);
2342         prval("tcpCurrEstab",           tcp->tcpCurrEstab);
2343         prval64("tcpOutSegs",           tcp->tcpHCOutSegs);
2344         prval("tcpOutDataSegs",         tcp->tcpOutDataSegs);
2345         prval("tcpOutDataBytes",        tcp->tcpOutDataBytes);
2346         prval("tcpRetransSegs",         tcp->tcpRetransSegs);
2347         prval("tcpRetransBytes",        tcp->tcpRetransBytes);
2348         prval("tcpOutAck",              tcp->tcpOutAck);
2349         prval("tcpOutAckDelayed",       tcp->tcpOutAckDelayed);
2350         prval("tcpOutUrg",              tcp->tcpOutUrg);
2351         prval("tcpOutWinUpdate",        tcp->tcpOutWinUpdate);
2352         prval("tcpOutWinProbe",         tcp->tcpOutWinProbe);
2353         prval("tcpOutControl",          tcp->tcpOutControl);
2354         prval("tcpOutRsts",             tcp->tcpOutRsts);
2355         prval("tcpOutFastRetrans",      tcp->tcpOutFastRetrans);
2356         prval64("tcpInSegs",            tcp->tcpHCInSegs);
2357         prval_end();
2358         prval("tcpInAckSegs",           tcp->tcpInAckSegs);
2359         prval("tcpInAckBytes",          tcp->tcpInAckBytes);
2360         prval("tcpInDupAck",            tcp->tcpInDupAck);
2361         prval("tcpInAckUnsent",         tcp->tcpInAckUnsent);
2362         prval("tcpInInorderSegs",       tcp->tcpInDataInorderSegs);
2363         prval("tcpInInorderBytes",      tcp->tcpInDataInorderBytes);
2364         prval("tcpInUnorderSegs",       tcp->tcpInDataUnorderSegs);
2365         prval("tcpInUnorderBytes",      tcp->tcpInDataUnorderBytes);
2366         prval("tcpInDupSegs",           tcp->tcpInDataDupSegs);
2367         prval("tcpInDupBytes",          tcp->tcpInDataDupBytes);
2368         prval("tcpInPartDupSegs",       tcp->tcpInDataPartDupSegs);
2369         prval("tcpInPartDupBytes",      tcp->tcpInDataPartDupBytes);
2370         prval("tcpInPastWinSegs",       tcp->tcpInDataPastWinSegs);
2371         prval("tcpInPastWinBytes",      tcp->tcpInDataPastWinBytes);
2372         prval("tcpInWinProbe",          tcp->tcpInWinProbe);
2373         prval("tcpInWinUpdate",         tcp->tcpInWinUpdate);
```

```
2374         prval("tcpInClosed",            tcp->tcpInClosed);
2375         prval("tcpRttNoUpdate",         tcp->tcpRttNoUpdate);
2376         prval("tcpRttUpdate",           tcp->tcpRttUpdate);
2377         prval("tcpTimRetrans",          tcp->tcpTimRetrans);
2378         prval("tcpTimRetransDrop",      tcp->tcpTimRetransDrop);
2379         prval("tcpTimKeepalive",        tcp->tcpTimKeepalive);
2380         prval("tcpTimKeepaliveProbe",   tcp->tcpTimKeepaliveProbe);
2381         prval("tcpTimKeepaliveDrop",    tcp->tcpTimKeepaliveDrop);
2382         prval("tcpListenDrop",          tcp->tcpListenDrop);
2383         prval("tcpListenDropQ0",        tcp->tcpListenDropQ0);
2384         prval("tcpHalfOpenDrop",        tcp->tcpHalfOpenDrop);
2385         prval("tcpOutSackRetrans",      tcp->tcpOutSackRetransSegs);
2386         prval_end();

2388 }

2390 static void
2391 print_udp_stats(mib2_udp_t *udp)
2392 {
2393         prval_init();
2394         prval64("udpInDatagrams",       udp->udpHCInDatagrams);
2395         prval("udpInErrors",            udp->udpInErrors);
2396         prval64("udpOutDatagrams",      udp->udpHCOutDatagrams);
2397         prval("udpOutErrors",           udp->udpOutErrors);
2398         prval_end();
2399 }

2401 static void
2402 print_rawip_stats(mib2_rawip_t *rawip)
2403 {
2404         prval_init();
2405         prval("rawipInDatagrams",       rawip->rawipInDatagrams);
2406         prval("rawipInErrors",          rawip->rawipInErrors);
2407         prval("rawipInCksumErrs",       rawip->rawipInCksumErrs);
2408         prval("rawipOutDatagrams",      rawip->rawipOutDatagrams);
2409         prval("rawipOutErrors",         rawip->rawipOutErrors);
2410         prval_end();
2411 }

2413 void
2414 print_igmp_stats(struct igmpstat *igps)
2415 {
2416         (void) printf(" %10u message%s received\n",
2417             igps->igps_rcv_total, PLURAL(igps->igps_rcv_total));
2418         (void) printf(" %10u message%s received with too few bytes\n",
2419             igps->igps_rcv_tooshort, PLURAL(igps->igps_rcv_tooshort));
2420         (void) printf(" %10u message%s received with bad checksum\n",
2421             igps->igps_rcv_badsum, PLURAL(igps->igps_rcv_badsum));
2422         (void) printf(" %10u membership quer%s received\n",
2423             igps->igps_rcv_queries, PLURALY(igps->igps_rcv_queries));
2424         (void) printf(" %10u membership quer%s received with invalid "
2425             "field(s)\n",
2426             igps->igps_rcv_badqueries, PLURALY(igps->igps_rcv_badqueries));
2427         (void) printf(" %10u membership report%s received\n",
2428             igps->igps_rcv_reports, PLURAL(igps->igps_rcv_reports));
2429         (void) printf(" %10u membership report%s received with invalid "
2430             "field(s)\n",
2431             igps->igps_rcv_badreports, PLURAL(igps->igps_rcv_badreports));
2432         (void) printf(" %10u membership report%s received for groups to "
2433             "which we belong\n",
2434             igps->igps_rcv_ourreports, PLURAL(igps->igps_rcv_ourreports));
2435         (void) printf(" %10u membership report%s sent\n",
2436             igps->igps_snd_reports, PLURAL(igps->igps_snd_reports));
2437 }

2439 static void
```



```
2440 print_mrt_stats(struct mrtstat *mrts)
2441 {
2442         (void) puts("DVMRP multicast routing:");
2443         (void) printf(" %10u hit%s - kernel forwarding cache hits\n",
2444             mrts->mrts_mfc_hits, PLURAL(mrts->mrts_mfc_hits));
2445         (void) printf(" %10u miss%s - kernel forwarding cache misses\n",
2446             mrts->mrts_mfc_misses, PLURALES(mrts->mrts_mfc_misses));
2447         (void) printf(" %10u packet%s potentially forwarded\n",
2448             mrts->mrts_fwd_in, PLURAL(mrts->mrts_fwd_in));
2449         (void) printf(" %10u packet%s actually sent out\n",
2450             mrts->mrts_fwd_out, PLURAL(mrts->mrts_fwd_out));
2451         (void) printf(" %10u upcall%s - upcalls made to mrouted\n",
2452             mrts->mrts_upcalls, PLURAL(mrts->mrts_upcalls));
2453         (void) printf(" %10u packet%s not sent out due to lack of resources\n",
2454             mrts->mrts_fwd_drop, PLURAL(mrts->mrts_fwd_drop));
2455         (void) printf(" %10u datagram%s with malformed tunnel options\n",
2456             mrts->mrts_bad_tunnel, PLURAL(mrts->mrts_bad_tunnel));
2457         (void) printf(" %10u datagram%s with no room for tunnel options\n",
2458             mrts->mrts_cant_tunnel, PLURAL(mrts->mrts_cant_tunnel));
2459         (void) printf(" %10u datagram%s arrived on wrong interface\n",
2460             mrts->mrts_wrong_if, PLURAL(mrts->mrts_wrong_if));
2461         (void) printf(" %10u datagram%s dropped due to upcall Q overflow\n",
2462             mrts->mrts_upq_ovflw, PLURAL(mrts->mrts_upq_ovflw));
2463         (void) printf(" %10u datagram%s cleaned up by the cache\n",
2464             mrts->mrts_cache_cleanups, PLURAL(mrts->mrts_cache_cleanups));
2465         (void) printf(" %10u datagram%s dropped selectively by ratelimiter\n",
2466             mrts->mrts_drop_sel, PLURAL(mrts->mrts_drop_sel));
2467         (void) printf(" %10u datagram%s dropped - bucket Q overflow\n",
2468             mrts->mrts_q_overflow, PLURAL(mrts->mrts_q_overflow));
2469         (void) printf(" %10u datagram%s dropped - larger than bkt size\n",
2470             mrts->mrts_pkt2large, PLURAL(mrts->mrts_pkt2large));
2471         (void) printf("\nPIM multicast routing:\n");
2472         (void) printf(" %10u datagram%s dropped - bad version number\n",
2473             mrts->mrts_pim_badversion, PLURAL(mrts->mrts_pim_badversion));
2474         (void) printf(" %10u datagram%s dropped - bad checksum\n",
2475             mrts->mrts_pim_rcv_badcsum, PLURAL(mrts->mrts_pim_rcv_badcsum));
2476         (void) printf(" %10u datagram%s dropped - bad register packets\n",
2477             mrts->mrts_pim_badregisters, PLURAL(mrts->mrts_pim_badregisters));
2478         (void) printf(
2479             " %10u datagram%s potentially forwarded - register packets\n",
2480             mrts->mrts_pim_regforwards, PLURAL(mrts->mrts_pim_regforwards));
2481         (void) printf(" %10u datagram%s dropped - register send drops\n",
2482             mrts->mrts_pim_regsend_drops, PLURAL(mrts->mrts_pim_regsend_drops));
2483         (void) printf(" %10u datagram%s dropped - packet malformed\n",
2484             mrts->mrts_pim_malformed, PLURAL(mrts->mrts_pim_malformed));
2485         (void) printf(" %10u datagram%s dropped - no memory to forward\n",
2486             mrts->mrts_pim_nomemory, PLURAL(mrts->mrts_pim_nomemory));
2487 }

2489 static void
2490 sum_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6, mib2_ipv6IfStatsEntry_t *sum6)
2491 {
2492         /* First few are not additive */
2493         sum6->ipv6Forwarding = ip6->ipv6Forwarding;
2494         sum6->ipv6DefaultHopLimit = ip6->ipv6DefaultHopLimit;

2496         sum6->ipv6InReceives += ip6->ipv6InReceives;
2497         sum6->ipv6InHdrErrors += ip6->ipv6InHdrErrors;
2498         sum6->ipv6InTooBigErrors += ip6->ipv6InTooBigErrors;
2499         sum6->ipv6InNoRoutes += ip6->ipv6InNoRoutes;
2500         sum6->ipv6InAddrErrors += ip6->ipv6InAddrErrors;
2501         sum6->ipv6InUnknownProtos += ip6->ipv6InUnknownProtos;
2502         sum6->ipv6InTruncatedPkts += ip6->ipv6InTruncatedPkts;
2503         sum6->ipv6InDiscards += ip6->ipv6InDiscards;
2504         sum6->ipv6InDelivers += ip6->ipv6InDelivers;
2505         sum6->ipv6OutForwDatagrams += ip6->ipv6OutForwDatagrams;
```

```
2506         sum6->ipv6OutRequests += ip6->ipv6OutRequests;
2507         sum6->ipv6OutDiscards += ip6->ipv6OutDiscards;
2508         sum6->ipv6OutFragOKs += ip6->ipv6OutFragOKs;
2509         sum6->ipv6OutFragFails += ip6->ipv6OutFragFails;
2510         sum6->ipv6OutFragCreates += ip6->ipv6OutFragCreates;
2511         sum6->ipv6ReasmReqds += ip6->ipv6ReasmReqds;
2512         sum6->ipv6ReasmOKs += ip6->ipv6ReasmOKs;
2513         sum6->ipv6ReasmFails += ip6->ipv6ReasmFails;
2514         sum6->ipv6InMcastPkts += ip6->ipv6InMcastPkts;
2515         sum6->ipv6OutMcastPkts += ip6->ipv6OutMcastPkts;
2516         sum6->ipv6OutNoRoutes += ip6->ipv6OutNoRoutes;
2517         sum6->ipv6ReasmDuplicates += ip6->ipv6ReasmDuplicates;
2518         sum6->ipv6ReasmPartDups += ip6->ipv6ReasmPartDups;
2519         sum6->ipv6ForwProhibits += ip6->ipv6ForwProhibits;
2520         sum6->udpInCksumErrs += ip6->udpInCksumErrs;
2521         sum6->udpInOverflows += ip6->udpInOverflows;
2522         sum6->rawipInOverflows += ip6->rawipInOverflows;
2523 }

2525 static void
2526 sum_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6, mib2_ipv6IfIcmpEntry_t *sum6)
2527 {
2528         sum6->ipv6IfIcmpInMsgs += icmp6->ipv6IfIcmpInMsgs;
2529         sum6->ipv6IfIcmpInErrors += icmp6->ipv6IfIcmpInErrors;
2530         sum6->ipv6IfIcmpInDestUnreachs += icmp6->ipv6IfIcmpInDestUnreachs;
2531         sum6->ipv6IfIcmpInAdminProhibs += icmp6->ipv6IfIcmpInAdminProhibs;
2532         sum6->ipv6IfIcmpInTimeExcds += icmp6->ipv6IfIcmpInTimeExcds;
2533         sum6->ipv6IfIcmpInParmProblems += icmp6->ipv6IfIcmpInParmProblems;
2534         sum6->ipv6IfIcmpInPktTooBigs += icmp6->ipv6IfIcmpInPktTooBigs;
2535         sum6->ipv6IfIcmpInEchos += icmp6->ipv6IfIcmpInEchos;
2536         sum6->ipv6IfIcmpInEchoReplies += icmp6->ipv6IfIcmpInEchoReplies;
2537         sum6->ipv6IfIcmpInRouterSolicits += icmp6->ipv6IfIcmpInRouterSolicits;
2538         sum6->ipv6IfIcmpInRouterAdvertisements +=
2539             icmp6->ipv6IfIcmpInRouterAdvertisements;
2540         sum6->ipv6IfIcmpInNeighborSolicits +=
2541             icmp6->ipv6IfIcmpInNeighborSolicits;
2542         sum6->ipv6IfIcmpInNeighborAdvertisements +=
2543             icmp6->ipv6IfIcmpInNeighborAdvertisements;
2544         sum6->ipv6IfIcmpInRedirects += icmp6->ipv6IfIcmpInRedirects;
2545         sum6->ipv6IfIcmpInGroupMembQueries +=
2546             icmp6->ipv6IfIcmpInGroupMembQueries;
2547         sum6->ipv6IfIcmpInGroupMembResponses +=
2548             icmp6->ipv6IfIcmpInGroupMembResponses;
2549         sum6->ipv6IfIcmpInGroupMembReductions +=
2550             icmp6->ipv6IfIcmpInGroupMembReductions;
2551         sum6->ipv6IfIcmpOutMsgs += icmp6->ipv6IfIcmpOutMsgs;
2552         sum6->ipv6IfIcmpOutErrors += icmp6->ipv6IfIcmpOutErrors;
2553         sum6->ipv6IfIcmpOutDestUnreachs += icmp6->ipv6IfIcmpOutDestUnreachs;
2554         sum6->ipv6IfIcmpOutAdminProhibs += icmp6->ipv6IfIcmpOutAdminProhibs;
2555         sum6->ipv6IfIcmpOutTimeExcds += icmp6->ipv6IfIcmpOutTimeExcds;
2556         sum6->ipv6IfIcmpOutParmProblems += icmp6->ipv6IfIcmpOutParmProblems;
2557         sum6->ipv6IfIcmpOutPktTooBigs += icmp6->ipv6IfIcmpOutPktTooBigs;
2558         sum6->ipv6IfIcmpOutEchos += icmp6->ipv6IfIcmpOutEchos;
2559         sum6->ipv6IfIcmpOutEchoReplies += icmp6->ipv6IfIcmpOutEchoReplies;
2560         sum6->ipv6IfIcmpOutRouterSolicits +=
2561             icmp6->ipv6IfIcmpOutRouterSolicits;
2562         sum6->ipv6IfIcmpOutRouterAdvertisements +=
2563             icmp6->ipv6IfIcmpOutRouterAdvertisements;
2564         sum6->ipv6IfIcmpOutNeighborSolicits +=
2565             icmp6->ipv6IfIcmpOutNeighborSolicits;
2566         sum6->ipv6IfIcmpOutNeighborAdvertisements +=
2567             icmp6->ipv6IfIcmpOutNeighborAdvertisements;
2568         sum6->ipv6IfIcmpOutRedirects += icmp6->ipv6IfIcmpOutRedirects;
2569         sum6->ipv6IfIcmpOutGroupMembQueries +=
2570             icmp6->ipv6IfIcmpOutGroupMembQueries;
2571         sum6->ipv6IfIcmpOutGroupMembResponses +=
```



```
2572             icmp6->ipv6IfIcmpOutGroupMembResponses;
2573         sum6->ipv6IfIcmpOutGroupMembReductions +=
2574             icmp6->ipv6IfIcmpOutGroupMembReductions;
2575         sum6->ipv6IfIcmpInOverflows += icmp6->ipv6IfIcmpInOverflows;
2576 }

2578 /* ---------------------------- MRT_STAT_REPORT -------------------------- */

2580 static void
2581 mrt_stat_report(mib_item_t *curritem)
2582 {
2583         int     jtemp = 0;
2584         mib_item_t *tempitem;

2586         if (!(family_selected(AF_INET)))
2587                 return;

2589         (void) putchar('\n');
2590         /* 'for' loop 1: */
2591         for (tempitem = curritem;
2592             tempitem;
2593             tempitem = tempitem->next_item) {
2594                 if (Xflag) {
2595                         (void) printf("\n--- Entry %d ---\n", ++jtemp);
2596                         (void) printf("Group = %d, mib_id = %d, "
2597                             "length = %d, valp = 0x%p\n",
2598                             tempitem->group, tempitem->mib_id,
2599                             tempitem->length, tempitem->valp);
2600                 }

2602                 if (tempitem->mib_id == 0) {
2603                         switch (tempitem->group) {
2604                         case EXPER_DVMRP: {
2605                                 struct mrtstat  *mrts;
2606                                 mrts = (struct mrtstat *)tempitem->valp;

2608                                 if (!(family_selected(AF_INET)))
2609                                         continue; /* 'for' loop 1 */

2611                                 print_mrt_stats(mrts);
2612                                 break;
2613                         }
2614                         }
2615                 }
2616         } /* 'for' loop 1 ends */
2617         (void) putchar('\n');
2618         (void) fflush(stdout);
2619 }

2621 /*
2622  * if_stat_total() - Computes totals for interface statistics
2623  *                   and returns result by updating sumstats.
2624  */
2625 static void
2626 if_stat_total(struct ifstat *oldstats, struct ifstat *newstats,
2627     struct ifstat *sumstats)
2628 {
2629         sumstats->ipackets += newstats->ipackets - oldstats->ipackets;
2630         sumstats->opackets += newstats->opackets - oldstats->opackets;
2631         sumstats->ierrors += newstats->ierrors - oldstats->ierrors;
2632         sumstats->oerrors += newstats->oerrors - oldstats->oerrors;
2633         sumstats->collisions += newstats->collisions - oldstats->collisions;
2634 }

2636 /* --------------------- IF_REPORT (netstat -i)  ------------------------- */
```

```
2638 static struct   ifstat  zerostat = {
2639         0LL, 0LL, 0LL, 0LL, 0LL
2640 };

2642 static void
2643 if_report(mib_item_t *item, char *matchname,
2644     int Iflag_only, boolean_t once_only)
2645 {
2646         static boolean_t        reentry = B_FALSE;
2647         boolean_t               alreadydone = B_FALSE;
2648         int                     jtemp = 0;
2649         uint32_t                ifindex_v4 = 0;
2650         uint32_t                ifindex_v6 = 0;
2651         boolean_t               first_header = B_TRUE;

2653         /* 'for' loop 1: */
2654         for (; item; item = item->next_item) {
2655                 if (Xflag) {
2656                         (void) printf("\n--- Entry %d ---\n", ++jtemp);
2657                         (void) printf("Group = %d, mib_id = %d, "
2658                             "length = %d, valp = 0x%p\n",
2659                             item->group, item->mib_id, item->length,
2660                             item->valp);
2661                 }

2663                 switch (item->group) {
2664                 case MIB2_IP:
2665                 if (item->mib_id != MIB2_IP_ADDR ||
2666                     !family_selected(AF_INET))
2667                         continue; /* 'for' loop 1 */
2668                 {
2669                         static struct ifstat    old = {0L, 0L, 0L, 0L, 0L};
2670                         static struct ifstat    new = {0L, 0L, 0L, 0L, 0L};
2671                         struct ifstat           sum;
2672                         struct iflist           *newlist = NULL;
2673                         static struct iflist    *oldlist = NULL;
2674                         kstat_t  *ksp;

2676                         if (once_only) {
2677                                 char    ifname[LIFNAMSIZ + 1];
2678                                 char    logintname[LIFNAMSIZ + 1];
2679                                 mib2_ipAddrEntry_t *ap;
2680                                 struct ifstat   stat = {0L, 0L, 0L, 0L, 0L};
2681                                 boolean_t       first = B_TRUE;
2682                                 uint32_t        new_ifindex;

2684                                 if (Xflag)
2685                                         (void) printf("if_report: %d items\n",
2686                                             (item->length)
2687                                             / sizeof (mib2_ipAddrEntry_t));

2689                                 /* 'for' loop 2a: */
2690                                 for (ap = (mib2_ipAddrEntry_t *)item->valp;
2691                                     (char *)ap < (char *)item->valp
2692                                     + item->length;
2693                                     ap++) {
2694                                         (void) octetstr(&ap->ipAdEntIfIndex,
2695                                             'a', logintname,
2696                                             sizeof (logintname));
2697                                         (void) strcpy(ifname, logintname);
2698                                         (void) strtok(ifname, ":");
2699                                         if (matchname != NULL &&
2700                                             strcmp(matchname, ifname) != 0 &&
2701                                             strcmp(matchname, logintname) != 0)
2702                                                 continue; /* 'for' loop 2a */
2703                                         new_ifindex =
```



```
2704                                             if_nametoindex(logintname);
2705                                         /*
2706                                          * First lookup the "link" kstats in
2707                                          * case the link is renamed. Then
2708                                          * fallback to the legacy kstats for
2709                                          * those non-GLDv3 links.
2710                                          */
2711                                         if (new_ifindex != ifindex_v4 &&
2712                                             (((ksp = kstat_lookup(kc, "link", 0,
2713                                             ifname)) != NULL) ||
2714                                             ((ksp = kstat_lookup(kc, NULL, -1,
2715                                             ifname)) != NULL))) {
2716                                                 (void) safe_kstat_read(kc, ksp,
2717                                                     NULL);
2718                                                 stat.ipackets =
2719                                                     kstat_named_value(ksp,
2720                                                     "ipackets");
2721                                                 stat.ierrors =
2722                                                     kstat_named_value(ksp,
2723                                                     "ierrors");
2724                                                 stat.opackets =
2725                                                     kstat_named_value(ksp,
2726                                                     "opackets");
2727                                                 stat.oerrors =
2728                                                     kstat_named_value(ksp,
2729                                                     "oerrors");
2730                                                 stat.collisions =
2731                                                     kstat_named_value(ksp,
2732                                                     "collisions");
2733                                                 if (first) {
2734                                                         if (!first_header)
2735                                                         (void) putchar('\n');
2736                                                         first_header = B_FALSE;
2737                                                 (void) printf(
2738                                                     "%-5.5s %-5.5s%-13.13s "
2739                                                     "%-14.14s %-6.6s %-5.5s "
2740                                                     "%-6.6s %-5.5s %-6.6s "
2741                                                     "%-6.6s\n",
2742                                                     "Name", "Mtu", "Net/Dest",
2743                                                     "Address", "Ipkts",
2744                                                     "Ierrs", "Opkts", "Oerrs",
2745                                                     "Collis", "Queue");

2747                                                 first = B_FALSE;
2748                                                 }
2749                                                 if_report_ip4(ap, ifname,
2750                                                     logintname, &stat, B_TRUE);
2751                                                 ifindex_v4 = new_ifindex;
2752                                         } else {
2753                                                 if_report_ip4(ap, ifname,
2754                                                     logintname, &stat, B_FALSE);
2755                                         }
2756                                 } /* 'for' loop 2a ends */
2757                         } else if (!alreadydone) {
2758                                 char    ifname[LIFNAMSIZ + 1];
2759                                 char    buf[LIFNAMSIZ + 1];
2760                                 mib2_ipAddrEntry_t *ap;
2761                                 struct ifstat   t;
2762                                 struct iflist   *tlp = NULL;
2763                                 struct iflist   **nextnew = &newlist;
2764                                 struct iflist   *walkold;
2765                                 struct iflist   *cleanlist;
2766                                 boolean_t       found_if = B_FALSE;

2768                                 alreadydone = B_TRUE; /* ignore other case */
```

```
2770                            /*
2771                             * Check if there is anything to do.
2772                             */
2773                            if (item->length <
2774                                sizeof (mib2_ipAddrEntry_t)) {
2775                                    fail(0, "No compatible interfaces");
2776                            }

2778                            /*
2779                             * 'for' loop 2b: find the "right" entry:
2780                             * If an interface name to match has been
2781                             * supplied then try and find it, otherwise
2782                             * match the first non-loopback interface found.
2783                             * Use lo0 if all else fails.
2784                             */
2785                            for (ap = (mib2_ipAddrEntry_t *)item->valp;
2786                                (char *)ap < (char *)item->valp
2787                                + item->length;
2788                                ap++) {
2789                                    (void) octetstr(&ap->ipAdEntIfIndex,
2790                                        'a', ifname, sizeof (ifname));
2791                                    (void) strtok(ifname, ":");

2793                                    if (matchname) {
2794                                            if (strcmp(matchname,
2795                                                ifname) == 0) {
2796                                                    /* 'for' loop 2b */
2797                                                    found_if = B_TRUE;
2798                                                    break;
2799                                            }
2800                                    } else if (strcmp(ifname, "lo0") != 0)
2801                                            break; /* 'for' loop 2b */
2802                            } /* 'for' loop 2b ends */

2804                            if (matchname == NULL) {
2805                                    matchname = ifname;
2806                            } else {
2807                                    if (!found_if)
2808                                            fail(0, "-I: %s no such "
2809                                                "interface.", matchname);
2810                            }

2812                            if (Iflag_only == 0 || !reentry) {
2813                                    (void) printf("    input   %-6.6s    "
2814                                        "output     ",
2815                                        matchname);
2816                                    (void) printf("   input  (Total)    "
2817                                    "output\n");
2818                                    (void) printf("%-7.7s %-5.5s %-7.7s "
2819                                        "%-5.5s %-6.6s ",
2820                                        "packets", "errs", "packets",
2821                                        "errs", "colls");
2822                                    (void) printf("%-7.7s %-5.5s %-7.7s "
2823                                        "%-5.5s %-6.6s\n",
2824                                        "packets", "errs", "packets",
2825                                        "errs", "colls");
2826                            }

2828                            sum = zerostat;

2830                            /* 'for' loop 2c: */
2831                            for (ap = (mib2_ipAddrEntry_t *)item->valp;
2832                                (char *)ap < (char *)item->valp
2833                                + item->length;
2834                                ap++) {
2835                                    (void) octetstr(&ap->ipAdEntIfIndex,
```



```
2836                                        'a', buf, sizeof (buf));
2837                                (void) strtok(buf, ":");

2839                                /*
2840                                 * We have reduced the IP interface
2841                                 * name, which could have been a
2842                                 * logical, down to a name suitable
2843                                 * for use with kstats.
2844                                 * We treat this name as unique and
2845                                 * only collate statistics for it once
2846                                 * per pass. This is to avoid falsely
2847                                 * amplifying these statistics by the
2848                                 * the number of logical instances.
2849                                 */
2850                                if ((tlp != NULL) &&
2851                                    ((strcmp(buf, tlp->ifname) == 0))) {
2852                                        continue;
2853                                }

2855                                /*
2856                                 * First lookup the "link" kstats in
2857                                 * case the link is renamed. Then
2858                                 * fallback to the legacy kstats for
2859                                 * those non-GLDv3 links.
2860                                 */
2861                                if (((ksp = kstat_lookup(kc, "link",
2862                                    0, buf)) != NULL ||
2863                                    (ksp = kstat_lookup(kc, NULL, -1,
2864                                    buf)) != NULL) && (ksp->ks_type ==
2865                                    KSTAT_TYPE_NAMED)) {
2866                                        (void) safe_kstat_read(kc, ksp,
2867                                            NULL);
2868                                }

2870                                t.ipackets = kstat_named_value(ksp,
2871                                    "ipackets");
2872                                t.ierrors = kstat_named_value(ksp,
2873                                    "ierrors");
2874                                t.opackets = kstat_named_value(ksp,
2875                                    "opackets");
2876                                t.oerrors = kstat_named_value(ksp,
2877                                    "oerrors");
2878                                t.collisions = kstat_named_value(ksp,
2879                                    "collisions");

2881                                if (strcmp(buf, matchname) == 0)
2882                                        new = t;

2884                                /* Build the interface list */

2886                                tlp = malloc(sizeof (struct iflist));
2887                                (void) strlcpy(tlp->ifname, buf,
2888                                    sizeof (tlp->ifname));
2889                                tlp->tot = t;
2890                                *nextnew = tlp;
2891                                nextnew = &tlp->next_if;

2893                                /*
2894                                 * First time through.
2895                                 * Just add up the interface stats.
2896                                 */

2898                                if (oldlist == NULL) {
2899                                        if_stat_total(&zerostat,
2900                                            &t, &sum);
2901                                        continue;
```

```
2902                                    }
2903
2904                                    /*
2905                                     * Walk old list for the interface.
2906                                     *
2907                                     * If found, add difference to total.
2908                                     *
2909                                     * If not, an interface has been plumbed
2910                                     * up.  In this case, we will simply
2911                                     * ignore the new interface until the
2912                                     * next interval; as there's no easy way
2913                                     * to acquire statistics between time
2914                                     * of the plumb and the next interval
2915                                     * boundary.  This results in inaccurate
2916                                     * total values for current interval.
2917                                     *
2918                                     * Note the case when an interface is
2919                                     * unplumbed; as similar problems exist.
2920                                     * The unplumbed interface is not in the
2921                                     * current list, and there's no easy way
2922                                     * to account for the statistics between
2923                                     * the previous interval and time of the
2924                                     * unplumb.  Therefore, we (in a sense)
2925                                     * ignore the removed interface by only
2926                                     * involving "current" interfaces when
2927                                     * computing the total statistics.
2928                                     * Unfortunately, this also results in
2929                                     * inaccurate values for interval total.
2930                                     */
2931
2932                                    for (walkold = oldlist;
2933                                        walkold != NULL;
2934                                        walkold = walkold->next_if) {
2935                                            if (strcmp(walkold->ifname,
2936                                                buf) == 0) {
2937                                                    if_stat_total(
2938                                                        &walkold->tot,
2939                                                        &t, &sum);
2940                                                    break;
2941                                            }
2942                                    }
2943
2944                                } /* 'for' loop 2c ends */
2945
2946                                *nextnew = NULL;
2947
2948                                (void) printf("%-7llu %-5llu %-7llu "
2949                                    "%-5llu %-6llu ",
2950                                    new.ipackets - old.ipackets,
2951                                    new.ierrors - old.ierrors,
2952                                    new.opackets - old.opackets,
2953                                    new.oerrors - old.oerrors,
2954                                    new.collisions - old.collisions);
2955
2956                                (void) printf("%-7llu %-5llu %-7llu "
2957                                    "%-5llu %-6llu\n", sum.ipackets,
2958                                    sum.ierrors, sum.opackets,
2959                                    sum.oerrors, sum.collisions);
2960
2961                                /*
2962                                 * Tidy things up once finished.
2963                                 */
2964
2965                                old = new;
2966                                cleanlist = oldlist;
2967                                oldlist = newlist;
```



```
2968                                    while (cleanlist != NULL) {
2969                                            tlp = cleanlist->next_if;
2970                                            free(cleanlist);
2971                                            cleanlist = tlp;
2972                                    }
2973                            }
2974                            break;
2975                    }
2976                    case MIB2_IP6:
2977                    if (item->mib_id != MIB2_IP6_ADDR ||
2978                        !family_selected(AF_INET6))
2979                            continue; /* 'for' loop 1 */
2980                    {
2981                            static struct ifstat    old6 = {0L, 0L, 0L, 0L, 0L};
2982                            static struct ifstat    new6 = {0L, 0L, 0L, 0L, 0L};
2983                            struct ifstat           sum6;
2984                            struct iflist           *newlist6 = NULL;
2985                            static struct iflist    *oldlist6 = NULL;
2986                            kstat_t  *ksp;
2987
2988                            if (once_only) {
2989                                    char    ifname[LIFNAMSIZ + 1];
2990                                    char    logintname[LIFNAMSIZ + 1];
2991                                    mib2_ipv6AddrEntry_t *ap6;
2992                                    struct ifstat   stat = {0L, 0L, 0L, 0L, 0L};
2993                                    boolean_t       first = B_TRUE;
2994                                    uint32_t        new_ifindex;
2995
2996                                    if (Xflag)
2997                                            (void) printf("if_report: %d items\n",
2998                                                (item->length)
2999                                                / sizeof (mib2_ipv6AddrEntry_t));
3000                                    /* 'for' loop 2d: */
3001                                    for (ap6 = (mib2_ipv6AddrEntry_t *)item->valp;
3002                                        (char *)ap6 < (char *)item->valp
3003                                        + item->length;
3004                                        ap6++) {
3005                                            (void) octetstr(&ap6->ipv6AddrIfIndex,
3006                                                'a', logintname,
3007                                                sizeof (logintname));
3008                                            (void) strcpy(ifname, logintname);
3009                                            (void) strtok(ifname, ":");
3010                                            if (matchname != NULL &&
3011                                                strcmp(matchname, ifname) != 0 &&
3012                                                strcmp(matchname, logintname) != 0)
3013                                                    continue; /* 'for' loop 2d */
3014                                            new_ifindex =
3015                                                if_nametoindex(logintname);
3016
3017                                            /*
3018                                             * First lookup the "link" kstats in
3019                                             * case the link is renamed. Then
3020                                             * fallback to the legacy kstats for
3021                                             * those non-GLDv3 links.
3022                                             */
3023                                            if (new_ifindex != ifindex_v6 &&
3024                                                ((ksp = kstat_lookup(kc, "link", 0,
3025                                                ifname)) != NULL ||
3026                                                (ksp = kstat_lookup(kc, NULL, -1,
3027                                                ifname)) != NULL)) {
3028                                                    (void) safe_kstat_read(kc, ksp,
3029                                                        NULL);
3030                                                    stat.ipackets =
3031                                                        kstat_named_value(ksp,
3032                                                        "ipackets");
3033                                                    stat.ierrors =
```

```
3034                                            kstat_named_value(ksp,
3035                                            "ierrors");
3036                                    stat.opackets =
3037                                            kstat_named_value(ksp,
3038                                            "opackets");
3039                                    stat.oerrors =
3040                                            kstat_named_value(ksp,
3041                                            "oerrors");
3042                                    stat.collisions =
3043                                            kstat_named_value(ksp,
3044                                            "collisions");
3045                                    if (first) {
3046                                            if (!first_header)
3047                                            (void) putchar('\n');
3048                                            first_header = B_FALSE;
3049                                            (void) printf(
3050                                                "%-5.5s %-5.5s%"
3051                                                "-27.27s %-27.27s "
3052                                                "%-6.6s %-5.5s "
3053                                                "%-6.6s %-5.5s "
3054                                                "%-6.6s\n",
3055                                                "Name", "Mtu",
3056                                                "Net/Dest",
3057                                                "Address", "Ipkts",
3058                                                "Ierrs", "Opkts",
3059                                                "Oerrs", "Collis");
3060                                            first = B_FALSE;
3061                                    }
3062                                    if_report_ip6(ap6, ifname,
3063                                        logintname, &stat, B_TRUE);
3064                                    ifindex_v6 = new_ifindex;
3065                            } else {
3066                                    if_report_ip6(ap6, ifname,
3067                                        logintname, &stat, B_FALSE);
3068                            }
3069                    } /* 'for' loop 2d ends */
3070            } else if (!alreadydone) {
3071                    char    ifname[LIFNAMSIZ + 1];
3072                    char    buf[IFNAMSIZ + 1];
3073                    mib2_ipv6AddrEntry_t *ap6;
3074                    struct ifstat   t;
3075                    struct iflist   *tlp = NULL;
3076                    struct iflist   **nextnew = &newlist6;
3077                    struct iflist   *walkold;
3078                    struct iflist   *cleanlist;
3079                    boolean_t       found_if = B_FALSE;

3081                    alreadydone = B_TRUE; /* ignore other case */

3083                    /*
3084                     * Check if there is anything to do.
3085                     */
3086                    if (item->length <
3087                        sizeof (mib2_ipv6AddrEntry_t)) {
3088                            fail(0, "No compatible interfaces");
3089                    }

3091                    /*
3092                     * 'for' loop 2e: find the "right" entry:
3093                     * If an interface name to match has been
3094                     * supplied then try and find it, otherwise
3095                     * match the first non-loopback interface found.
3096                     * Use lo0 if all else fails.
3097                     */
3098                    for (ap6 = (mib2_ipv6AddrEntry_t *)item->valp;
3099                        (char *)ap6 < (char *)item->valp
```



```
3100                                    + item->length;
3101                                    ap6++) {
3102                                        (void) octetstr(&ap6->ipv6AddrIfIndex,
3103                                            'a', ifname, sizeof (ifname));
3104                                        (void) strtok(ifname, ":");

3106                                        if (matchname) {
3107                                                if (strcmp(matchname,
3108                                                    ifname) == 0) {
3109                                                    /* 'for' loop 2e */
3110                                                    found_if = B_TRUE;
3111                                                    break;
3112                                                }
3113                                        } else if (strcmp(ifname, "lo0") != 0)
3114                                                break; /* 'for' loop 2e */
3115                            } /* 'for' loop 2e ends */

3117                            if (matchname == NULL) {
3118                                    matchname = ifname;
3119                            } else {
3120                                    if (!found_if)
3121                                            fail(0, "-I: %s no such "
3122                                                "interface.", matchname);
3123                            }

3125                            if (Iflag_only == 0 || !reentry) {
3126                                    (void) printf(
3127                                        "    input   %-6.6s"
3128                                        "    output ",
3129                                        matchname);
3130                                    (void) printf("   input  (Total)"
3131                                        "    output\n");
3132                                    (void) printf("%-7.7s %-5.5s %-7.7s "
3133                                        "%-5.5s %-6.6s ",
3134                                        "packets", "errs", "packets",
3135                                        "errs", "colls");
3136                                    (void) printf("%-7.7s %-5.5s %-7.7s "
3137                                        "%-5.5s %-6.6s\n",
3138                                        "packets", "errs", "packets",
3139                                        "errs", "colls");
3140                            }

3142                            sum6 = zerostat;

3144                            /* 'for' loop 2f: */
3145                            for (ap6 = (mib2_ipv6AddrEntry_t *)item->valp;
3146                                (char *)ap6 < (char *)item->valp
3147                                + item->length;
3148                                ap6++) {
3149                                    (void) octetstr(&ap6->ipv6AddrIfIndex,
3150                                        'a', buf, sizeof (buf));
3151                                    (void) strtok(buf, ":");

3153                                    /*
3154                                     * We have reduced the IP interface
3155                                     * name, which could have been a
3156                                     * logical, down to a name suitable
3157                                     * for use with kstats.
3158                                     * We treat this name as unique and
3159                                     * only collate statistics for it once
3160                                     * per pass. This is to avoid falsely
3161                                     * amplifying these statistics by the
3162                                     * the number of logical instances.
3163                                     */

3165                                    if ((tlp != NULL) &&
```

```
3166                                        ((strcmp(buf, tlp->ifname) == 0))) {
3167                                            continue;
3168                                    }

3170                                    /*
3171                                     * First lookup the "link" kstats in
3172                                     * case the link is renamed. Then
3173                                     * fallback to the legacy kstats for
3174                                     * those non-GLDv3 links.
3175                                     */
3176                                    if (((ksp = kstat_lookup(kc, "link",
3177                                        0, buf)) != NULL ||
3178                                        (ksp = kstat_lookup(kc, NULL, -1,
3179                                        buf)) != NULL) && (ksp->ks_type ==
3180                                        KSTAT_TYPE_NAMED)) {
3181                                            (void) safe_kstat_read(kc,
3182                                                ksp, NULL);
3183                                    }

3185                                    t.ipackets = kstat_named_value(ksp,
3186                                        "ipackets");
3187                                    t.ierrors = kstat_named_value(ksp,
3188                                        "ierrors");
3189                                    t.opackets = kstat_named_value(ksp,
3190                                        "opackets");
3191                                    t.oerrors = kstat_named_value(ksp,
3192                                        "oerrors");
3193                                    t.collisions = kstat_named_value(ksp,
3194                                        "collisions");

3196                                    if (strcmp(buf, matchname) == 0)
3197                                            new6 = t;

3199                                    /* Build the interface list */

3201                                    tlp = malloc(sizeof (struct iflist));
3202                                    (void) strlcpy(tlp->ifname, buf,
3203                                        sizeof (tlp->ifname));
3204                                    tlp->tot = t;
3205                                    *nextnew = tlp;
3206                                    nextnew = &tlp->next_if;

3208                                    /*
3209                                     * First time through.
3210                                     * Just add up the interface stats.
3211                                     */

3213                                    if (oldlist6 == NULL) {
3214                                            if_stat_total(&zerostat,
3215                                                &t, &sum6);
3216                                            continue;
3217                                    }

3219                                    /*
3220                                     * Walk old list for the interface.
3221                                     *
3222                                     * If found, add difference to total.
3223                                     *
3224                                     * If not, an interface has been plumbed
3225                                     * up.  In this case, we will simply
3226                                     * ignore the new interface until the
3227                                     * next interval; as there's no easy way
3228                                     * to acquire statistics between time
3229                                     * of the plumb and the next interval
3230                                     * boundary.  This results in inaccurate
3231                                     * total values for current interval.
```



```
3232                                     *
3233                                     * Note the case when an interface is
3234                                     * unplumbed; as similar problems exist.
3235                                     * The unplumbed interface is not in the
3236                                     * current list, and there's no easy way
3237                                     * to account for the statistics between
3238                                     * the previous interval and time of the
3239                                     * unplumb.  Therefore, we (in a sense)
3240                                     * ignore the removed interface by only
3241                                     * involving "current" interfaces when
3242                                     * computing the total statistics.
3243                                     * Unfortunately, this also results in
3244                                     * inaccurate values for interval total.
3245                                     */

3247                                    for (walkold = oldlist6;
3248                                        walkold != NULL;
3249                                        walkold = walkold->next_if) {
3250                                            if (strcmp(walkold->ifname,
3251                                                buf) == 0) {
3252                                                if_stat_total(
3253                                                    &walkold->tot,
3254                                                    &t, &sum6);
3255                                                break;
3256                                            }
3257                                    }

3259                                } /* 'for' loop 2f ends */

3261                                *nextnew = NULL;

3263                                (void) printf("%-7llu %-5llu %-7llu "
3264                                    "%-5llu %-6llu ",
3265                                    new6.ipackets - old6.ipackets,
3266                                    new6.ierrors - old6.ierrors,
3267                                    new6.opackets - old6.opackets,
3268                                    new6.oerrors - old6.oerrors,
3269                                    new6.collisions - old6.collisions);

3271                                (void) printf("%-7llu %-5llu %-7llu "
3272                                    "%-5llu %-6llu\n", sum6.ipackets,
3273                                    sum6.ierrors, sum6.opackets,
3274                                    sum6.oerrors, sum6.collisions);

3276                                /*
3277                                 * Tidy things up once finished.
3278                                 */

3280                                old6 = new6;
3281                                cleanlist = oldlist6;
3282                                oldlist6 = newlist6;
3283                                while (cleanlist != NULL) {
3284                                        tlp = cleanlist->next_if;
3285                                        free(cleanlist);
3286                                        cleanlist = tlp;
3287                                }
3288                        }
3289                        break;
3290                }
3291                }
3292                (void) fflush(stdout);
3293        } /* 'for' loop 1 ends */
3294        if ((Iflag_only == 0) && (!once_only))
3295                (void) putchar('\n');
3296        reentry = B_TRUE;
3297 }
```

```
3299 static void
3300 if_report_ip4(mib2_ipAddrEntry_t *ap,
3301         char ifname[], char logintname[], struct ifstat *statptr,
3302         boolean_t ksp_not_null) {

3304         char abuf[MAXHOSTNAMELEN + 1];
3305         char dstbuf[MAXHOSTNAMELEN + 1];

3307         if (ksp_not_null) {
3308                 (void) printf("%-5s %-4u ",
3309                     ifname, ap->ipAdEntInfo.ae_mtu);
3310                 if (ap->ipAdEntInfo.ae_flags & IFF_POINTOPOINT)
3311                         (void) pr_addr(ap->ipAdEntInfo.ae_pp_dst_addr,
3312                             abuf, sizeof (abuf));
3313                 else
3314                         (void) pr_netaddr(ap->ipAdEntAddr,
3315                             ap->ipAdEntNetMask, abuf, sizeof (abuf));
3316                 (void) printf("%-13s %-14s %-6llu %-5llu %-6llu %-5llu "
3317                     "%-6llu %-6llu\n",
3318                     abuf, pr_addr(ap->ipAdEntAddr, dstbuf, sizeof (dstbuf)),
3319                     statptr->ipackets, statptr->ierrors,
3320                     statptr->opackets, statptr->oerrors,
3321                     statptr->collisions, 0LL);
3322         }
3323         /*
3324          * Print logical interface info if Aflag set (including logical unit 0)
3325          */
3326         if (Aflag) {
3327                 *statptr = zerostat;
3328                 statptr->ipackets = ap->ipAdEntInfo.ae_ibcnt;
3329                 statptr->opackets = ap->ipAdEntInfo.ae_obcnt;

3331                 (void) printf("%-5s %-4u ", logintname, ap->ipAdEntInfo.ae_mtu);
3332                 if (ap->ipAdEntInfo.ae_flags & IFF_POINTOPOINT)
3333                         (void) pr_addr(ap->ipAdEntInfo.ae_pp_dst_addr, abuf,
3334                         sizeof (abuf));
3335                 else
3336                         (void) pr_netaddr(ap->ipAdEntAddr, ap->ipAdEntNetMask,
3337                             abuf, sizeof (abuf));

3339                 (void) printf("%-13s %-14s %-6llu %-5s %-6s "
3340                     "%-5s %-6s %-6llu\n", abuf,
3341                     pr_addr(ap->ipAdEntAddr, dstbuf, sizeof (dstbuf)),
3342                     statptr->ipackets, "N/A", "N/A", "N/A", "N/A",
3343                     0LL);
3344         }
3345 }

3347 static void
3348 if_report_ip6(mib2_ipv6AddrEntry_t *ap6,
3349         char ifname[], char logintname[], struct ifstat *statptr,
3350         boolean_t ksp_not_null) {

3352         char abuf[MAXHOSTNAMELEN + 1];
3353         char dstbuf[MAXHOSTNAMELEN + 1];

3355         if (ksp_not_null) {
3356                 (void) printf("%-5s %-4u ", ifname, ap6->ipv6AddrInfo.ae_mtu);
3357                 if (ap6->ipv6AddrInfo.ae_flags &
3358                     IFF_POINTOPOINT) {
3359                         (void) pr_addr6(&ap6->ipv6AddrInfo.ae_pp_dst_addr,
3360                             abuf, sizeof (abuf));
3361                 } else {
3362                         (void) pr_prefix6(&ap6->ipv6AddrAddress,
3363                             ap6->ipv6AddrPfxLength, abuf,
```



```
3364                             sizeof (abuf));
3365                 }
3366                 (void) printf("%-27s %-27s %-6llu %-5llu "
3367                     "%-6llu %-5llu %-6llu\n",
3368                     abuf, pr_addr6(&ap6->ipv6AddrAddress, dstbuf,
3369                     sizeof (dstbuf)),
3370                     statptr->ipackets, statptr->ierrors, statptr->opackets,
3371                     statptr->oerrors, statptr->collisions);
3372         }
3373         /*
3374          * Print logical interface info if Aflag set (including logical unit 0)
3375          */
3376         if (Aflag) {
3377                 *statptr = zerostat;
3378                 statptr->ipackets = ap6->ipv6AddrInfo.ae_ibcnt;
3379                 statptr->opackets = ap6->ipv6AddrInfo.ae_obcnt;

3381                 (void) printf("%-5s %-4u ", logintname,
3382                     ap6->ipv6AddrInfo.ae_mtu);
3383                 if (ap6->ipv6AddrInfo.ae_flags & IFF_POINTOPOINT)
3384                         (void) pr_addr6(&ap6->ipv6AddrInfo.ae_pp_dst_addr,
3385                             abuf, sizeof (abuf));
3386                 else
3387                         (void) pr_prefix6(&ap6->ipv6AddrAddress,
3388                             ap6->ipv6AddrPfxLength, abuf, sizeof (abuf));
3389                 (void) printf("%-27s %-27s %-6llu %-5s %-6s %-5s %-6s\n",
3390                     abuf, pr_addr6(&ap6->ipv6AddrAddress, dstbuf,
3391                     sizeof (dstbuf)),
3392                     statptr->ipackets, "N/A", "N/A", "N/A", "N/A");
3393         }
3394 }

3396 /* -------------------- DHCP_REPORT  (netstat -D) ------------------------ */

3398 static boolean_t
3399 dhcp_do_ipc(dhcp_ipc_type_t type, const char *ifname, boolean_t printed_one)
3400 {
3401         dhcp_ipc_request_t      *request;
3402         dhcp_ipc_reply_t        *reply;
3403         int                     error;

3405         request = dhcp_ipc_alloc_request(type, ifname, NULL, 0, DHCP_TYPE_NONE);
3406         if (request == NULL)
3407                 fail(0, "dhcp_do_ipc: out of memory");

3409         error = dhcp_ipc_make_request(request, &reply, DHCP_IPC_WAIT_DEFAULT);
3410         if (error != 0) {
3411                 free(request);
3412                 fail(0, "dhcp_do_ipc: %s", dhcp_ipc_strerror(error));
3413         }

3415         free(request);
3416         error = reply->return_code;
3417         if (error == DHCP_IPC_E_UNKIF) {
3418                 free(reply);
3419                 return (printed_one);
3420         }
3421         if (error != 0) {
3422                 free(reply);
3423                 fail(0, "dhcp_do_ipc: %s", dhcp_ipc_strerror(error));
3424         }

3426         if (timestamp_fmt != NODATE)
3427                 print_timestamp(timestamp_fmt);

3429         if (!printed_one)
```

```
3430                (void) printf("%s", dhcp_status_hdr_string());

3432        (void) printf("%s", dhcp_status_reply_to_string(reply));
3433        free(reply);
3434        return (B_TRUE);
3435 }

3437 /*
3438  * dhcp_walk_interfaces: walk the list of interfaces for a given address
3439  * family (af).  For each, print out the DHCP status using dhcp_do_ipc.
3440  */
3441 static boolean_t
3442 dhcp_walk_interfaces(int af, boolean_t printed_one)
3443 {
3444        struct lifnum   lifn;
3445        struct lifconf  lifc;
3446        int             n_ifs, i, sock_fd;

3448        sock_fd = socket(af, SOCK_DGRAM, 0);
3449        if (sock_fd == -1)
3450                return (printed_one);

3452        /*
3453         * SIOCGLIFNUM is just an estimate.  If the ioctl fails, we don't care;
3454         * just drive on and use SIOCGLIFCONF with increasing buffer sizes, as
3455         * is traditional.
3456         */
3457        (void) memset(&lifn, 0, sizeof (lifn));
3458        lifn.lifn_family = af;
3459        lifn.lifn_flags = LIFC_ALLZONES | LIFC_NOXMIT | LIFC_UNDER_IPMP;
3460        if (ioctl(sock_fd, SIOCGLIFNUM, &lifn) == -1)
3461                n_ifs = LIFN_GUARD_VALUE;
3462        else
3463                n_ifs = lifn.lifn_count + LIFN_GUARD_VALUE;

3465        (void) memset(&lifc, 0, sizeof (lifc));
3466        lifc.lifc_family = af;
3467        lifc.lifc_flags = lifn.lifn_flags;
3468        lifc.lifc_len = n_ifs * sizeof (struct lifreq);
3469        lifc.lifc_buf = malloc(lifc.lifc_len);
3470        if (lifc.lifc_buf != NULL) {

3472                if (ioctl(sock_fd, SIOCGLIFCONF, &lifc) == -1) {
3473                        (void) close(sock_fd);
3474                        free(lifc.lifc_buf);
3475                        return (NULL);
3476                }

3478                n_ifs = lifc.lifc_len / sizeof (struct lifreq);

3480                for (i = 0; i < n_ifs; i++) {
3481                        printed_one = dhcp_do_ipc(DHCP_STATUS |
3482                            (af == AF_INET6 ? DHCP_V6 : 0),
3483                            lifc.lifc_req[i].lifr_name, printed_one);
3484                }
3485        }
3486        (void) close(sock_fd);
3487        free(lifc.lifc_buf);
3488        return (printed_one);
3489 }

3491 static void
3492 dhcp_report(char *ifname)
3493 {
3494        boolean_t printed_one;
```



```
3496        if (!family_selected(AF_INET) && !family_selected(AF_INET6))
3497                return;

3499        printed_one = B_FALSE;
3500        if (ifname != NULL) {
3501                if (family_selected(AF_INET)) {
3502                        printed_one = dhcp_do_ipc(DHCP_STATUS, ifname,
3503                            printed_one);
3504                }
3505                if (family_selected(AF_INET6)) {
3506                        printed_one = dhcp_do_ipc(DHCP_STATUS | DHCP_V6,
3507                            ifname, printed_one);
3508                }
3509                if (!printed_one) {
3510                        fail(0, "%s: %s", ifname,
3511                            dhcp_ipc_strerror(DHCP_IPC_E_UNKIF));
3512                }
3513        } else {
3514                if (family_selected(AF_INET)) {
3515                        printed_one = dhcp_walk_interfaces(AF_INET,
3516                            printed_one);
3517                }
3518                if (family_selected(AF_INET6))
3519                        (void) dhcp_walk_interfaces(AF_INET6, printed_one);
3520        }
3521 }

3523 /* --------------------- GROUP_REPORT (netstat -g) ------------------------ */

3525 static void
3526 group_report(mib_item_t *item)
3527 {
3528        mib_item_t      *v4grp = NULL, *v4src = NULL;
3529        mib_item_t      *v6grp = NULL, *v6src = NULL;
3530        int             jtemp = 0;
3531        char            ifname[LIFNAMSIZ + 1];
3532        char            abuf[MAXHOSTNAMELEN + 1];
3533        ip_member_t     *ipmp;
3534        ip_grpsrc_t     *ips;
3535        ipv6_member_t   *ipmp6;
3536        ipv6_grpsrc_t   *ips6;
3537        boolean_t       first, first_src;

3539        /* 'for' loop 1: */
3540        for (; item; item = item->next_item) {
3541                if (Xflag) {
3542                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
3543                        (void) printf("Group = %d, mib_id = %d, "
3544                            "length = %d, valp = 0x%p\n",
3545                            item->group, item->mib_id, item->length,
3546                            item->valp);
3547                }
3548                if (item->group == MIB2_IP && family_selected(AF_INET)) {
3549                        switch (item->mib_id) {
3550                        case EXPER_IP_GROUP_MEMBERSHIP:
3551                                v4grp = item;
3552                                if (Xflag)
3553                                        (void) printf("item is v4grp info\n");
3554                                break;
3555                        case EXPER_IP_GROUP_SOURCES:
3556                                v4src = item;
3557                                if (Xflag)
3558                                        (void) printf("item is v4src info\n");
3559                                break;
3560                        default:
3561                                continue;
```

```
3562                        }
3563                        continue;
3564                }
3565                if (item->group == MIB2_IP6 && family_selected(AF_INET6)) {
3566                        switch (item->mib_id) {
3567                        case EXPER_IP6_GROUP_MEMBERSHIP:
3568                                v6grp = item;
3569                                if (Xflag)
3570                                        (void) printf("item is v6grp info\n");
3571                                break;
3572                        case EXPER_IP6_GROUP_SOURCES:
3573                                v6src = item;
3574                                if (Xflag)
3575                                        (void) printf("item is v6src info\n");
3576                                break;
3577                        default:
3578                                continue;
3579                        }
3580                }
3581        }

3583        if (family_selected(AF_INET) && v4grp != NULL) {
3584                if (Xflag)
3585                        (void) printf("%u records for ipGroupMember:\n",
3586                            v4grp->length / sizeof (ip_member_t));

3588                first = B_TRUE;
3589                for (ipmp = (ip_member_t *)v4grp->valp;
3590                    (char *)ipmp < (char *)v4grp->valp + v4grp->length;
3591                    /* LINTED: (note 1) */
3592                    ipmp = (ip_member_t *)((char *)ipmp + ipMemberEntrySize)) {
3593                        if (first) {
3594                                (void) puts(v4compat ?
3595                                    "Group Memberships" :
3596                                    "Group Memberships: IPv4");
3597                                (void) puts("Interface "
3598                                    "Group              RefCnt");
3599                                (void) puts("--------- "
3600                                    "------------------- ------");
3601                                first = B_FALSE;
3602                        }

3604                        (void) printf("%-9s %-20s %6u\n",
3605                            octetstr(&ipmp->ipGroupMemberIfIndex, 'a',
3606                            ifname, sizeof (ifname)),
3607                            pr_addr(ipmp->ipGroupMemberAddress,
3608                            abuf, sizeof (abuf)),
3609                            ipmp->ipGroupMemberRefCnt);


3612                        if (!Vflag || v4src == NULL)
3613                                continue;

3615                        if (Xflag)
3616                                (void) printf("scanning %u ipGroupSource "
3617                                    "records...\n",
3618                                    v4src->length/sizeof (ip_grpsrc_t));

3620                        first_src = B_TRUE;
3621                        for (ips = (ip_grpsrc_t *)v4src->valp;
3622                            (char *)ips < (char *)v4src->valp + v4src->length;
3623                            /* LINTED: (note 1) */
3624                            ips = (ip_grpsrc_t *)((char *)ips +
3625                            ipGroupSourceEntrySize)) {
3626                                /*
3627                                 * We assume that all source addrs for a given
```



```
3628                                 * interface/group pair are contiguous, so on
3629                                 * the first non-match after we've found at
3630                                 * least one, we bail.
3631                                 */
3632                                if ((ipmp->ipGroupMemberAddress !=
3633                                    ips->ipGroupSourceGroup) ||
3634                                    (!octetstrmatch(&ipmp->ipGroupMemberIfIndex,
3635                                    &ips->ipGroupSourceIfIndex))) {
3636                                        if (first_src)
3637                                                continue;
3638                                        else
3639                                                break;
3640                                }
3641                                if (first_src) {
3642                                        (void) printf("\t%s:    %s\n",
3643                                            fmodestr(
3644                                            ipmp->ipGroupMemberFilterMode),
3645                                            pr_addr(ips->ipGroupSourceAddress,
3646                                            abuf, sizeof (abuf)));
3647                                        first_src = B_FALSE;
3648                                        continue;
3649                                }

3651                                (void) printf("\t            %s\n",
3652                                    pr_addr(ips->ipGroupSourceAddress, abuf,
3653                                    sizeof (abuf)));
3654                        }
3655                }
3656                (void) putchar('\n');
3657        }

3659        if (family_selected(AF_INET6) && v6grp != NULL) {
3660                if (Xflag)
3661                        (void) printf("%u records for ipv6GroupMember:\n",
3662                            v6grp->length / sizeof (ipv6_member_t));

3664                first = B_TRUE;
3665                for (ipmp6 = (ipv6_member_t *)v6grp->valp;
3666                    (char *)ipmp6 < (char *)v6grp->valp + v6grp->length;
3667                    /* LINTED: (note 1) */
3668                    ipmp6 = (ipv6_member_t *)((char *)ipmp6 +
3669                    ipv6MemberEntrySize)) {
3670                        if (first) {
3671                                (void) puts("Group Memberships: "
3672                                    "IPv6");
3673                                (void) puts(" If       "
3674                                    "Group                    RefCnt");
3675                                (void) puts("----- "
3676                                    "--------------------------- ------");
3677                                first = B_FALSE;
3678                        }

3680                        (void) printf("%-5s %-27s %5u\n",
3681                            ifindex2str(ipmp6->ipv6GroupMemberIfIndex, ifname),
3682                            pr_addr6(&ipmp6->ipv6GroupMemberAddress,
3683                            abuf, sizeof (abuf)),
3684                            ipmp6->ipv6GroupMemberRefCnt);

3686                        if (!Vflag || v6src == NULL)
3687                                continue;

3689                        if (Xflag)
3690                                (void) printf("scanning %u ipv6GroupSource "
3691                                    "records...\n",
3692                                    v6src->length/sizeof (ipv6_grpsrc_t));
```

```
3694                        first_src = B_TRUE;
3695                        for (ips6 = (ipv6_grpsrc_t *)v6src->valp;
3696                            (char *)ips6 < (char *)v6src->valp + v6src->length;
3697                            /* LINTED: (note 1) */
3698                            ips6 = (ipv6_grpsrc_t *)((char *)ips6 +
3699                            ipv6GroupSourceEntrySize)) {
3700                                /* same assumption as in the v4 case above */
3701                                if ((ipmp6->ipv6GroupMemberIfIndex !=
3702                                    ips6->ipv6GroupSourceIfIndex) ||
3703                                    (!IN6_ARE_ADDR_EQUAL(
3704                                    &ipmp6->ipv6GroupMemberAddress,
3705                                    &ips6->ipv6GroupSourceGroup))) {
3706                                        if (first_src)
3707                                                continue;
3708                                        else
3709                                                break;
3710                                }
3711                                if (first_src) {
3712                                        (void) printf("\t%s:    %s\n",
3713                                            fmodestr(
3714                                            ipmp6->ipv6GroupMemberFilterMode),
3715                                            pr_addr6(
3716                                            &ips6->ipv6GroupSourceAddress,
3717                                            abuf, sizeof (abuf)));
3718                                        first_src = B_FALSE;
3719                                        continue;
3720                                }

3722                                (void) printf("\t           %s\n",
3723                                    pr_addr6(&ips6->ipv6GroupSourceAddress,
3724                                    abuf, sizeof (abuf)));
3725                        }
3726                }
3727                (void) putchar('\n');
3728        }

3730        (void) putchar('\n');
3731        (void) fflush(stdout);
3732 }

3734 /* --------------------- DCE_REPORT (netstat -d) ------------------------ */

3736 #define FLBUFSIZE       8

3738 /* Assumes flbuf is at least 5 characters; callers use FLBUFSIZE */
3739 static char *
3740 dceflags2str(uint32_t flags, char *flbuf)
3741 {
3742        char *str = flbuf;

3744        if (flags & DCEF_DEFAULT)
3745                *str++ = 'D';
3746        if (flags & DCEF_PMTU)
3747                *str++ = 'P';
3748        if (flags & DCEF_UINFO)
3749                *str++ = 'U';
3750        if (flags & DCEF_TOO_SMALL_PMTU)
3751                *str++ = 'S';
3752        *str++ = '\0';
3753        return (flbuf);
3754 }

3756 static void
3757 dce_report(mib_item_t *item)
3758 {
3759        mib_item_t      *v4dce = NULL;
```



```
3760        mib_item_t      *v6dce = NULL;
3761        int             jtemp = 0;
3762        char            ifname[LIFNAMSIZ + 1];
3763        char            abuf[MAXHOSTNAMELEN + 1];
3764        char            flbuf[FLBUFSIZE];
3765        boolean_t       first;
3766        dest_cache_entry_t *dce;

3768        /* 'for' loop 1: */
3769        for (; item; item = item->next_item) {
3770                if (Xflag) {
3771                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
3772                        (void) printf("Group = %d, mib_id = %d, "
3773                            "length = %d, valp = 0x%p\n",
3774                            item->group, item->mib_id, item->length,
3775                            item->valp);
3776                }
3777                if (item->group == MIB2_IP && family_selected(AF_INET) &&
3778                    item->mib_id == EXPER_IP_DCE) {
3779                        v4dce = item;
3780                        if (Xflag)
3781                                (void) printf("item is v4dce info\n");
3782                }
3783                if (item->group == MIB2_IP6 && family_selected(AF_INET6) &&
3784                    item->mib_id == EXPER_IP_DCE) {
3785                        v6dce = item;
3786                        if (Xflag)
3787                                (void) printf("item is v6dce info\n");
3788                }
3789        }

3791        if (family_selected(AF_INET) && v4dce != NULL) {
3792                if (Xflag)
3793                        (void) printf("%u records for DestCacheEntry:\n",
3794                            v4dce->length / ipDestEntrySize);

3796                first = B_TRUE;
3797                for (dce = (dest_cache_entry_t *)v4dce->valp;
3798                    (char *)dce < (char *)v4dce->valp + v4dce->length;
3799                    /* LINTED: (note 1) */
3800                    dce = (dest_cache_entry_t *)((char *)dce +
3801                    ipDestEntrySize)) {
3802                        if (first) {
3803                                (void) putchar('\n');
3804                                (void) puts("Destination Cache Entries: IPv4");
3805                                (void) puts(
3806                                    "Address               PMTU   Age  Flags");
3807                                (void) puts(
3808                                    "-------------------- ------ ----- -----");
3809                                first = B_FALSE;
3810                        }

3812                        (void) printf("%-20s %6u %5u %-5s\n",
3813                            pr_addr(dce->DestIpv4Address, abuf, sizeof (abuf)),
3814                            dce->DestPmtu, dce->DestAge,
3815                            dceflags2str(dce->DestFlags, flbuf));
3816                }
3817        }

3819        if (family_selected(AF_INET6) && v6dce != NULL) {
3820                if (Xflag)
3821                        (void) printf("%u records for DestCacheEntry:\n",
3822                            v6dce->length / ipDestEntrySize);

3824                first = B_TRUE;
3825                for (dce = (dest_cache_entry_t *)v6dce->valp;
```

```
3826                    (char *)dce < (char *)v6dce->valp + v6dce->length;
3827                    /* LINTED: (note 1) */
3828                    dce = (dest_cache_entry_t *)((char *)dce +
3829                    ipDestEntrySize)) {
3830                        if (first) {
3831                                (void) putchar('\n');
3832                                (void) puts("Destination Cache Entries: IPv6");
3833                                (void) puts(
3834                                    "Address                       PMTU  "
3835                                    " Age Flags If ");
3836                                (void) puts(
3837                                    "--------------------------- ------ "
3838                                    "----- ----- ---");
3839                                first = B_FALSE;
3840                        }

3842                        (void) printf("%-27s %6u %5u %-5s %s\n",
3843                            pr_addr6(&dce->DestIpv6Address, abuf,
3844                            sizeof (abuf)),
3845                            dce->DestPmtu, dce->DestAge,
3846                            dceflags2str(dce->DestFlags, flbuf),
3847                            dce->DestIfindex == 0 ? "" :
3848                            ifindex2str(dce->DestIfindex, ifname));
3849                }
3850        }
3851        (void) fflush(stdout);
3852 }

3854 /* -------------------- ARP_REPORT (netstat -p) ------------------------- */

3856 static void
3857 arp_report(mib_item_t *item)
3858 {
3859        int             jtemp = 0;
3860        char            ifname[LIFNAMSIZ + 1];
3861        char            abuf[MAXHOSTNAMELEN + 1];
3862        char            maskbuf[STR_EXPAND * OCTET_LENGTH + 1];
3863        char            flbuf[32];      /* ACE_F_ flags */
3864        char            xbuf[STR_EXPAND * OCTET_LENGTH + 1];
3865        mib2_ipNetToMediaEntry_t        *np;
3866        int             flags;
3867        boolean_t       first;

3869        if (!(family_selected(AF_INET)))
3870                return;

3872        /* 'for' loop 1: */
3873        for (; item; item = item->next_item) {
3874                if (Xflag) {
3875                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
3876                        (void) printf("Group = %d, mib_id = %d, "
3877                            "length = %d, valp = 0x%p\n",
3878                            item->group, item->mib_id, item->length,
3879                            item->valp);
3880                }
3881                if (!(item->group == MIB2_IP && item->mib_id == MIB2_IP_MEDIA))
3882                        continue; /* 'for' loop 1 */

3884                if (Xflag)
3885                        (void) printf("%u records for "
3886                            "ipNetToMediaEntryTable:\n",
3887                            item->length/sizeof (mib2_ipNetToMediaEntry_t));

3889                first = B_TRUE;
3890                /* 'for' loop 2: */
3891                for (np = (mib2_ipNetToMediaEntry_t *)item->valp;
```



```
3892                    (char *)np < (char *)item->valp + item->length;
3893                    /* LINTED: (note 1) */
3894                    np = (mib2_ipNetToMediaEntry_t *)((char *)np +
3895                    ipNetToMediaEntrySize)) {
3896                        if (first) {
3897                                (void) puts(v4compat ?
3898                                    "Net to Media Table" :
3899                                    "Net to Media Table: IPv4");
3900                                (void) puts("Device "
3901                                    "  IP Address               Mask      "
3902                                    "Flags      Phys Addr");
3903                                (void) puts("------ "
3904                                    "-------------------- --------------- "
3905                                    "-------- ---------------");
3906                                first = B_FALSE;
3907                        }

3909                        flbuf[0] = '\0';
3910                        flags = np->ipNetToMediaInfo.ntm_flags;
3911                        /*
3912                         * Note that not all flags are possible at the same
3913                         * time.  Patterns: SPLAy DUo
3914                         */
3915                        if (flags & ACE_F_PERMANENT)
3916                                (void) strcat(flbuf, "S");
3917                        if (flags & ACE_F_PUBLISH)
3918                                (void) strcat(flbuf, "P");
3919                        if (flags & ACE_F_DYING)
3920                                (void) strcat(flbuf, "D");
3921                        if (!(flags & ACE_F_RESOLVED))
3922                                (void) strcat(flbuf, "U");
3923                        if (flags & ACE_F_MAPPING)
3924                                (void) strcat(flbuf, "M");
3925                        if (flags & ACE_F_MYADDR)
3926                                (void) strcat(flbuf, "L");
3927                        if (flags & ACE_F_UNVERIFIED)
3928                                (void) strcat(flbuf, "d");
3929                        if (flags & ACE_F_AUTHORITY)
3930                                (void) strcat(flbuf, "A");
3931                        if (flags & ACE_F_OLD)
3932                                (void) strcat(flbuf, "o");
3933                        if (flags & ACE_F_DELAYED)
3934                                (void) strcat(flbuf, "y");
3935                        (void) printf("%-6s %-20s %-15s %-8s %s\n",
3936                            octetstr(&np->ipNetToMediaIfIndex, 'a',
3937                            ifname, sizeof (ifname)),
3938                            pr_addr(np->ipNetToMediaNetAddress,
3939                            abuf, sizeof (abuf)),
3940                            octetstr(&np->ipNetToMediaInfo.ntm_mask, 'd',
3941                            maskbuf, sizeof (maskbuf)),
3942                            flbuf,
3943                            octetstr(&np->ipNetToMediaPhysAddress, 'h',
3944                            xbuf, sizeof (xbuf)));
3945                } /* 'for' loop 2 ends */
3946        } /* 'for' loop 1 ends */
3947        (void) fflush(stdout);
3948 }

3950 /* -------------------- NDP_REPORT (netstat -p) ------------------------- */

3952 static void
3953 ndp_report(mib_item_t *item)
3954 {
3955        int             jtemp = 0;
3956        char            abuf[MAXHOSTNAMELEN + 1];
3957        char            *state;
```

```
3958        char            *type;
3959        char            xbuf[STR_EXPAND * OCTET_LENGTH + 1];
3960        mib2_ipv6NetToMediaEntry_t      *np6;
3961        char            ifname[LIFNAMSIZ + 1];
3962        boolean_t       first;

3964        if (!(family_selected(AF_INET6)))
3965                return;

3967        /* 'for' loop 1: */
3968        for (; item; item = item->next_item) {
3969                if (Xflag) {
3970                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
3971                        (void) printf("Group = %d, mib_id = %d, "
3972                            "length = %d, valp = 0x%p\n",
3973                            item->group, item->mib_id, item->length,
3974                            item->valp);
3975                }
3976                if (!(item->group == MIB2_IP6 &&
3977                    item->mib_id == MIB2_IP6_MEDIA))
3978                        continue; /* 'for' loop 1 */

3980                first = B_TRUE;
3981                /* 'for' loop 2: */
3982                for (np6 = (mib2_ipv6NetToMediaEntry_t *)item->valp;
3983                    (char *)np6 < (char *)item->valp + item->length;
3984                    /* LINTED: (note 1) */
3985                    np6 = (mib2_ipv6NetToMediaEntry_t *)((char *)np6 +
3986                    ipv6NetToMediaEntrySize)) {
3987                        if (first) {
3988                                (void) puts("\nNet to Media Table: IPv6");
3989                                (void) puts(" If   Physical Address   "
3990                                    " Type      State      Destination/Mask");
3991                                (void) puts("----- -----------------  "
3992                                    "------- ------------ "
3993                                    "--------------------------");
3994                                first = B_FALSE;
3995                        }

3997                        switch (np6->ipv6NetToMediaState) {
3998                        case ND_INCOMPLETE:
3999                                state = "INCOMPLETE";
4000                                break;
4001                        case ND_REACHABLE:
4002                                state = "REACHABLE";
4003                                break;
4004                        case ND_STALE:
4005                                state = "STALE";
4006                                break;
4007                        case ND_DELAY:
4008                                state = "DELAY";
4009                                break;
4010                        case ND_PROBE:
4011                                state = "PROBE";
4012                                break;
4013                        case ND_UNREACHABLE:
4014                                state = "UNREACHABLE";
4015                                break;
4016                        default:
4017                                state = "UNKNOWN";
4018                        }

4020                        switch (np6->ipv6NetToMediaType) {
4021                        case 1:
4022                                type = "other";
4023                                break;
```



```
4024                        case 2:
4025                                type = "dynamic";
4026                                break;
4027                        case 3:
4028                                type = "static";
4029                                break;
4030                        case 4:
4031                                type = "local";
4032                                break;
4033                        }
4034                        (void) printf("%-5s %-17s  %-7s %-12s %-27s\n",
4035                            ifindex2str(np6->ipv6NetToMediaIfIndex, ifname),
4036                            octetstr(&np6->ipv6NetToMediaPhysAddress, 'h',
4037                            xbuf, sizeof (xbuf)),
4038                            type,
4039                            state,
4040                            pr_addr6(&np6->ipv6NetToMediaNetAddress,
4041                            abuf, sizeof (abuf)));
4042                } /* 'for' loop 2 ends */
4043        } /* 'for' loop 1 ends */
4044        (void) putchar('\n');
4045        (void) fflush(stdout);
4046 }

4048 /* ------------------------ ire_report (netstat -r) ---------------------- */

4050 typedef struct sec_attr_list_s {
4051        struct sec_attr_list_s *sal_next;
4052        const mib2_ipAttributeEntry_t *sal_attr;
4053 } sec_attr_list_t;

4055 static boolean_t ire_report_item_v4(const mib2_ipRouteEntry_t *, boolean_t,
4056     const sec_attr_list_t *);
4057 static boolean_t ire_report_item_v6(const mib2_ipv6RouteEntry_t *, boolean_t,
4058     const sec_attr_list_t *);
4059 static const char *pr_secattr(const sec_attr_list_t *);

4061 static void
4062 ire_report(const mib_item_t *item)
4063 {
4064        int                     jtemp = 0;
4065        boolean_t               print_hdr_once_v4 = B_TRUE;
4066        boolean_t               print_hdr_once_v6 = B_TRUE;
4067        mib2_ipRouteEntry_t     *rp;
4068        mib2_ipv6RouteEntry_t   *rp6;
4069        sec_attr_list_t         **v4_attrs, **v4a;
4070        sec_attr_list_t         **v6_attrs, **v6a;
4071        sec_attr_list_t         *all_attrs, *aptr;
4072        const mib_item_t        *iptr;
4073        int                     ipv4_route_count, ipv6_route_count;
4074        int                     route_attrs_count;

4076        /*
4077         * Preparation pass: the kernel returns separate entries for IP routing
4078         * table entries and security attributes.  We loop through the
4079         * attributes first and link them into lists.
4080         */
4081        ipv4_route_count = ipv6_route_count = route_attrs_count = 0;
4082        for (iptr = item; iptr != NULL; iptr = iptr->next_item) {
4083                if (iptr->group == MIB2_IP6 && iptr->mib_id == MIB2_IP6_ROUTE)
4084                        ipv6_route_count += iptr->length / ipv6RouteEntrySize;
4085                if (iptr->group == MIB2_IP && iptr->mib_id == MIB2_IP_ROUTE)
4086                        ipv4_route_count += iptr->length / ipRouteEntrySize;
4087                if ((iptr->group == MIB2_IP || iptr->group == MIB2_IP6) &&
4088                    iptr->mib_id == EXPER_IP_RTATTR)
4089                        route_attrs_count += iptr->length /
```

```
4090                            ipRouteAttributeSize;
4091        }
4092        v4_attrs = v6_attrs = NULL;
4093        all_attrs = NULL;
4094        if (family_selected(AF_INET) && ipv4_route_count > 0) {
4095                v4_attrs = calloc(ipv4_route_count, sizeof (*v4_attrs));
4096                if (v4_attrs == NULL) {
4097                        perror("ire_report calloc v4_attrs failed");
4098                        return;
4099                }
4100        }
4101        if (family_selected(AF_INET6) && ipv6_route_count > 0) {
4102                v6_attrs = calloc(ipv6_route_count, sizeof (*v6_attrs));
4103                if (v6_attrs == NULL) {
4104                        perror("ire_report calloc v6_attrs failed");
4105                        goto ire_report_done;
4106                }
4107        }
4108        if (route_attrs_count > 0) {
4109                all_attrs = malloc(route_attrs_count * sizeof (*all_attrs));
4110                if (all_attrs == NULL) {
4111                        perror("ire_report malloc all_attrs failed");
4112                        goto ire_report_done;
4113                }
4114        }
4115        aptr = all_attrs;
4116        for (iptr = item; iptr != NULL; iptr = iptr->next_item) {
4117                mib2_ipAttributeEntry_t *iae;
4118                sec_attr_list_t **alp;

4120                if (v4_attrs != NULL && iptr->group == MIB2_IP &&
4121                    iptr->mib_id == EXPER_IP_RTATTR) {
4122                        alp = v4_attrs;
4123                } else if (v6_attrs != NULL && iptr->group == MIB2_IP6 &&
4124                    iptr->mib_id == EXPER_IP_RTATTR) {
4125                        alp = v6_attrs;
4126                } else {
4127                        continue;
4128                }
4129                for (iae = iptr->valp;
4130                    (char *)iae < (char *)iptr->valp + iptr->length;
4131                    /* LINTED: (note 1) */
4132                    iae = (mib2_ipAttributeEntry_t *)((char *)iae +
4133                    ipRouteAttributeSize)) {
4134                        aptr->sal_next = alp[iae->iae_routeidx];
4135                        aptr->sal_attr = iae;
4136                        alp[iae->iae_routeidx] = aptr++;
4137                }
4138        }

4140        /* 'for' loop 1: */
4141        v4a = v4_attrs;
4142        v6a = v6_attrs;
4143        for (; item != NULL; item = item->next_item) {
4144                if (Xflag) {
4145                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
4146                        (void) printf("Group = %d, mib_id = %d, "
4147                            "length = %d, valp = 0x%p\n",
4148                            item->group, item->mib_id,
4149                            item->length, item->valp);
4150                }
4151                if (!((item->group == MIB2_IP &&
4152                    item->mib_id == MIB2_IP_ROUTE) ||
4153                    (item->group == MIB2_IP6 &&
4154                    item->mib_id == MIB2_IP6_ROUTE)))
4155                        continue; /* 'for' loop 1 */
```



```
4157                if (item->group == MIB2_IP && !family_selected(AF_INET))
4158                        continue; /* 'for' loop 1 */
4159                else if (item->group == MIB2_IP6 && !family_selected(AF_INET6))
4160                        continue; /* 'for' loop 1 */

4162                if (Xflag) {
4163                        if (item->group == MIB2_IP) {
4164                                (void) printf("%u records for "
4165                                    "ipRouteEntryTable:\n",
4166                                    item->length/sizeof (mib2_ipRouteEntry_t));
4167                        } else {
4168                                (void) printf("%u records for "
4169                                    "ipv6RouteEntryTable:\n",
4170                                    item->length/
4171                                    sizeof (mib2_ipv6RouteEntry_t));
4172                        }
4173                }

4175                if (item->group == MIB2_IP) {
4176                        for (rp = (mib2_ipRouteEntry_t *)item->valp;
4177                            (char *)rp < (char *)item->valp + item->length;
4178                            /* LINTED: (note 1) */
4179                            rp = (mib2_ipRouteEntry_t *)((char *)rp +
4180                            ipRouteEntrySize)) {
4181                                aptr = v4a == NULL ? NULL : *v4a++;
4182                                print_hdr_once_v4 = ire_report_item_v4(rp,
4183                                    print_hdr_once_v4, aptr);
4184                        }
4185                } else {
4186                        for (rp6 = (mib2_ipv6RouteEntry_t *)item->valp;
4187                            (char *)rp6 < (char *)item->valp + item->length;
4188                            /* LINTED: (note 1) */
4189                            rp6 = (mib2_ipv6RouteEntry_t *)((char *)rp6 +
4190                            ipv6RouteEntrySize)) {
4191                                aptr = v6a == NULL ? NULL : *v6a++;
4192                                print_hdr_once_v6 = ire_report_item_v6(rp6,
4193                                    print_hdr_once_v6, aptr);
4194                        }
4195                }
4196        } /* 'for' loop 1 ends */
4197        (void) fflush(stdout);
4198 ire_report_done:
4199        if (v4_attrs != NULL)
4200                free(v4_attrs);
4201        if (v6_attrs != NULL)
4202                free(v6_attrs);
4203        if (all_attrs != NULL)
4204                free(all_attrs);
4205 }

4207 /*
4208  * Match a user-supplied device name.  We do this by string because
4209  * the MIB2 interface gives us interface name strings rather than
4210  * ifIndex numbers.  The "none" rule matches only routes with no
4211  * interface.  The "any" rule matches routes with any non-blank
4212  * interface.  A base name ("hme0") matches all aliases as well
4213  * ("hme0:1").
4214  */
4215 static boolean_t
4216 dev_name_match(const DeviceName *devnam, const char *ifname)
4217 {
4218        int iflen;

4220        if (ifname == NULL)
4221                return (devnam->o_length == 0);         /* "none" */
```

```
4222         if (*ifname == '\0')
4223                 return (devnam->o_length != 0);         /* "any" */
4224         iflen = strlen(ifname);
4225         /* The check for ':' here supports interface aliases. */
4226         if (iflen > devnam->o_length ||
4227             (iflen < devnam->o_length && devnam->o_bytes[iflen] != ':'))
4228                 return (B_FALSE);
4229         return (strncmp(ifname, devnam->o_bytes, iflen) == 0);
4230 }

4232 /*
4233  * Match a user-supplied IP address list.  The "any" rule matches any
4234  * non-zero address.  The "none" rule matches only the zero address.
4235  * IPv6 addresses supplied by the user are ignored.  If the user
4236  * supplies a subnet mask, then match routes that are at least that
4237  * specific (use the user's mask).  If the user supplies only an
4238  * address, then select any routes that would match (use the route's
4239  * mask).
4240  */
4241 static boolean_t
4242 v4_addr_match(IpAddress addr, IpAddress mask, const filter_t *fp)
4243 {
4244         char **app;
4245         char *aptr;
4246         in_addr_t faddr, fmask;

4248         if (fp->u.a.f_address == NULL) {
4249                 if (IN6_IS_ADDR_UNSPECIFIED(&fp->u.a.f_mask))
4250                         return (addr != INADDR_ANY);    /* "any" */
4251                 else
4252                         return (addr == INADDR_ANY);    /* "none" */
4253         }
4254         if (!IN6_IS_V4MASK(fp->u.a.f_mask))
4255                 return (B_FALSE);
4256         IN6_V4MAPPED_TO_IPADDR(&fp->u.a.f_mask, fmask);
4257         if (fmask != IP_HOST_MASK) {
4258                 if (fmask > mask)
4259                         return (B_FALSE);
4260                 mask = fmask;
4261         }
4262         for (app = fp->u.a.f_address->h_addr_list; (aptr = *app) != NULL; app++)
4263                 /* LINTED: (note 1) */
4264                 if (IN6_IS_ADDR_V4MAPPED((in6_addr_t *)aptr)) {
4265                         /* LINTED: (note 1) */
4266                         IN6_V4MAPPED_TO_IPADDR((in6_addr_t *)aptr, faddr);
4267                         if (((faddr ^ addr) & mask) == 0)
4268                                 return (B_TRUE);
4269                 }
4270         return (B_FALSE);
4271 }

4273 /*
4274  * Run through the filter list for an IPv4 MIB2 route entry.  If all
4275  * filters of a given type fail to match, then the route is filtered
4276  * out (not displayed).  If no filter is given or at least one filter
4277  * of each type matches, then display the route.
4278  */
4279 static boolean_t
4280 ire_filter_match_v4(const mib2_ipRouteEntry_t *rp, uint_t flag_b)
4281 {
4282         filter_t *fp;
4283         int idx;

4285         /* 'for' loop 1: */
4286         for (idx = 0; idx < NFILTERKEYS; idx++)
4287                 if ((fp = filters[idx]) != NULL) {
```



```
4288                         /* 'for' loop 2: */
4289                         for (; fp != NULL; fp = fp->f_next) {
4290                                 switch (idx) {
4291                                 case FK_AF:
4292                                         if (fp->u.f_family != AF_INET)
4293                                                 continue; /* 'for' loop 2 */
4294                                         break;
4295                                 case FK_OUTIF:
4296                                         if (!dev_name_match(&rp->ipRouteIfIndex,
4297                                             fp->u.f_ifname))
4298                                                 continue; /* 'for' loop 2 */
4299                                         break;
4300                                 case FK_DST:
4301                                         if (!v4_addr_match(rp->ipRouteDest,
4302                                             rp->ipRouteMask, fp))
4303                                                 continue; /* 'for' loop 2 */
4304                                         break;
4305                                 case FK_FLAGS:
4306                                         if ((flag_b & fp->u.f.f_flagset) !=
4307                                             fp->u.f.f_flagset ||
4308                                             (flag_b & fp->u.f.f_flagclear))
4309                                                 continue; /* 'for' loop 2 */
4310                                         break;
4311                                 }
4312                                 break;
4313                         } /* 'for' loop 2 ends */
4314                         if (fp == NULL)
4315                                 return (B_FALSE);
4316                 }
4317         /* 'for' loop 1 ends */
4318         return (B_TRUE);
4319 }

4321 /*
4322  * Given an IPv4 MIB2 route entry, form the list of flags for the
4323  * route.
4324  */
4325 static uint_t
4326 form_v4_route_flags(const mib2_ipRouteEntry_t *rp, char *flags)
4327 {
4328         uint_t flag_b;

4330         flag_b = FLF_U;
4331         (void) strcpy(flags, "U");
4332         /* RTF_INDIRECT wins over RTF_GATEWAY - don't display both */
4333         if (rp->ipRouteInfo.re_flags & RTF_INDIRECT) {
4334                 (void) strcat(flags, "I");
4335                 flag_b |= FLF_I;
4336         } else if (rp->ipRouteInfo.re_ire_type & IRE_OFFLINK) {
4337                 (void) strcat(flags, "G");
4338                 flag_b |= FLF_G;
4339         }
4340         /* IRE_IF_CLONE wins over RTF_HOST - don't display both */
4341         if (rp->ipRouteInfo.re_ire_type & IRE_IF_CLONE) {
4342                 (void) strcat(flags, "C");
4343                 flag_b |= FLF_C;
4344         } else if (rp->ipRouteMask == IP_HOST_MASK) {
4345                 (void) strcat(flags, "H");
4346                 flag_b |= FLF_H;
4347         }
4348         if (rp->ipRouteInfo.re_flags & RTF_DYNAMIC) {
4349                 (void) strcat(flags, "D");
4350                 flag_b |= FLF_D;
4351         }
4352         if (rp->ipRouteInfo.re_ire_type == IRE_BROADCAST) {     /* Broadcast */
4353                 (void) strcat(flags, "b");
```

```
4354                flag_b |= FLF_b;
4355        }
4356        if (rp->ipRouteInfo.re_ire_type == IRE_LOCAL) {         /* Local */
4357                (void) strcat(flags, "L");
4358                flag_b |= FLF_L;
4359        }
4360        if (rp->ipRouteInfo.re_flags & RTF_MULTIRT) {
4361                (void) strcat(flags, "M");                      /* Multiroute */
4362                flag_b |= FLF_M;
4363        }
4364        if (rp->ipRouteInfo.re_flags & RTF_SETSRC) {
4365                (void) strcat(flags, "S");                      /* Setsrc */
4366                flag_b |= FLF_S;
4367        }
4368        if (rp->ipRouteInfo.re_flags & RTF_REJECT) {
4369                (void) strcat(flags, "R");
4370                flag_b |= FLF_R;
4371        }
4372        if (rp->ipRouteInfo.re_flags & RTF_BLACKHOLE) {
4373                (void) strcat(flags, "B");
4374                flag_b |= FLF_B;
4375        }
4376        if (rp->ipRouteInfo.re_flags & RTF_ZONE) {
4377                (void) strcat(flags, "Z");
4378                flag_b |= FLF_Z;
4379        }
4380        return (flag_b);
4381 }

4383 static const char ire_hdr_v4[] =
4384 "\n%s Table: IPv4\n";
4385 static const char ire_hdr_v4_compat[] =
4386 "\n%s Table:\n";
4387 static const char ire_hdr_v4_verbose[] =
4388 "  Destination             Mask           Gateway          Device "
4389 " MTU  Ref Flg  Out  In/Fwd %s\n"
4390 "-------------------- --------------- -------------------- ------ "
4391 "----- --- --- ----- ------ %s\n";

4393 static const char ire_hdr_v4_normal[] =
4394 "  Destination           Gateway           Flags  Ref     Use     Interface"
4395 " %s\n-------------------- -------------------- ----- ----- ---------- "
4396 "--------- %s\n";

4398 static boolean_t
4399 ire_report_item_v4(const mib2_ipRouteEntry_t *rp, boolean_t first,
4400     const sec_attr_list_t *attrs)
4401 {
4402        char                    dstbuf[MAXHOSTNAMELEN + 1];
4403        char                    maskbuf[MAXHOSTNAMELEN + 1];
4404        char                    gwbuf[MAXHOSTNAMELEN + 1];
4405        char                    ifname[LIFNAMSIZ + 1];
4406        char                    flags[10];      /* RTF_ flags */
4407        uint_t                  flag_b;

4409        if (!(Aflag || (rp->ipRouteInfo.re_ire_type != IRE_IF_CLONE &&
4410            rp->ipRouteInfo.re_ire_type != IRE_BROADCAST &&
4411            rp->ipRouteInfo.re_ire_type != IRE_MULTICAST &&
4412            rp->ipRouteInfo.re_ire_type != IRE_NOROUTE &&
4413            rp->ipRouteInfo.re_ire_type != IRE_LOCAL))) {
4414                return (first);
4415        }

4417        flag_b = form_v4_route_flags(rp, flags);

4419        if (!ire_filter_match_v4(rp, flag_b))
```



```
4420                return (first);

4422        if (first) {
4423                (void) printf(v4compat ? ire_hdr_v4_compat : ire_hdr_v4,
4424                    Vflag ? "IRE" : "Routing");
4425                (void) printf(Vflag ? ire_hdr_v4_verbose : ire_hdr_v4_normal,
4426                    RSECflag ? "  Gateway security attributes  " : "",
4427                    RSECflag ? "-------------------------------" : "");
4428                first = B_FALSE;
4429        }

4431        if (flag_b & FLF_H) {
4432                (void) pr_addr(rp->ipRouteDest, dstbuf, sizeof (dstbuf));
4433        } else {
4434                (void) pr_net(rp->ipRouteDest, rp->ipRouteMask,
4435                    dstbuf, sizeof (dstbuf));
4436        }
4437        if (Vflag) {
4438                (void) printf("%-20s %-15s %-20s %-6s %5u %3u "
4439                    "%-4s%6u %6u %s\n",
4440                    dstbuf,
4441                    pr_mask(rp->ipRouteMask, maskbuf, sizeof (maskbuf)),
4442                    pr_addrnz(rp->ipRouteNextHop, gwbuf, sizeof (gwbuf)),
4443                    octetstr(&rp->ipRouteIfIndex, 'a', ifname, sizeof (ifname)),
4444                    rp->ipRouteInfo.re_max_frag,
4445                    rp->ipRouteInfo.re_ref,
4446                    flags,
4447                    rp->ipRouteInfo.re_obpkt,
4448                    rp->ipRouteInfo.re_ibpkt,
4449                    pr_secattr(attrs));
4450        } else {
4451                (void) printf("%-20s %-20s %-5s  %4u %10u %-9s %s\n",
4452                    dstbuf,
4453                    pr_addrnz(rp->ipRouteNextHop, gwbuf, sizeof (gwbuf)),
4454                    flags,
4455                    rp->ipRouteInfo.re_ref,
4456                    rp->ipRouteInfo.re_obpkt + rp->ipRouteInfo.re_ibpkt,
4457                    octetstr(&rp->ipRouteIfIndex, 'a',
4458                    ifname, sizeof (ifname)),
4459                    pr_secattr(attrs));
4460        }
4461        return (first);
4462 }

4464 /*
4465  * Match a user-supplied IP address list against an IPv6 route entry.
4466  * If the user specified "any," then any non-zero address matches.  If
4467  * the user specified "none," then only the zero address matches.  If
4468  * the user specified a subnet mask length, then use that in matching
4469  * routes (select routes that are at least as specific).  If the user
4470  * specified only an address, then use the route's mask (select routes
4471  * that would match that address).  IPv4 addresses are ignored.
4472  */
4473 static boolean_t
4474 v6_addr_match(const Ip6Address *addr, int masklen, const filter_t *fp)
4475 {
4476        const uint8_t *ucp;
4477        int fmasklen;
4478        int i;
4479        char **app;
4480        const uint8_t *aptr;

4482        if (fp->u.a.f_address == NULL) {
4483                if (IN6_IS_ADDR_UNSPECIFIED(&fp->u.a.f_mask))   /* any */
4484                        return (!IN6_IS_ADDR_UNSPECIFIED(addr));
4485                return (IN6_IS_ADDR_UNSPECIFIED(addr));         /* "none" */
```

```
4486            }
4487            fmasklen = 0;
4488            /* 'for' loop 1a: */
4489            for (ucp = fp->u.a.f_mask.s6_addr;
4490                ucp < fp->u.a.f_mask.s6_addr + sizeof (fp->u.a.f_mask.s6_addr);
4491                ucp++) {
4492                    if (*ucp != 0xff) {
4493                            if (*ucp != 0)
4494                                    fmasklen += 9 - ffs(*ucp);
4495                            break; /* 'for' loop 1a */
4496                    }
4497                    fmasklen += 8;
4498            } /* 'for' loop 1a ends */
4499            if (fmasklen != IPV6_ABITS) {
4500                    if (fmasklen > masklen)
4501                            return (B_FALSE);
4502                    masklen = fmasklen;
4503            }
4504            /* 'for' loop 1b: */
4505            for (app = fp->u.a.f_address->h_addr_list;
4506                (aptr = (uint8_t *)*app) != NULL; app++) {
4507                    /* LINTED: (note 1) */
4508                    if (IN6_IS_ADDR_V4MAPPED((in6_addr_t *)aptr))
4509                            continue; /* 'for' loop 1b */
4510                    ucp = addr->s6_addr;
4511                    for (i = masklen; i >= 8; i -= 8)
4512                            if (*ucp++ != *aptr++)
4513                                    break; /* 'for' loop 1b */
4514                    if (i == 0 ||
4515                        (i < 8 && ((*ucp ^ *aptr) & ~(0xff >> i)) == 0))
4516                            return (B_TRUE);
4517            } /* 'for' loop 1b ends */
4518            return (B_FALSE);
4519    }

4521    /*
4522     * Run through the filter list for an IPv6 MIB2 IRE.  For a given
4523     * type, if there's at least one filter and all filters of that type
4524     * fail to match, then the route doesn't match and isn't displayed.
4525     * If at least one matches, or none are specified, for each of the
4526     * types, then the route is selected and displayed.
4527     */
4528    static boolean_t
4529    ire_filter_match_v6(const mib2_ipv6RouteEntry_t *rp6, uint_t flag_b)
4530    {
4531            filter_t *fp;
4532            int idx;

4534            /* 'for' loop 1: */
4535            for (idx = 0; idx < NFILTERKEYS; idx++)
4536                    if ((fp = filters[idx]) != NULL) {
4537                            /* 'for' loop 2: */
4538                            for (; fp != NULL; fp = fp->f_next) {
4539                                    switch (idx) {
4540                                    case FK_AF:
4541                                            if (fp->u.f_family != AF_INET6)
4542                                                    /* 'for' loop 2 */
4543                                                    continue;
4544                                            break;
4545                                    case FK_OUTIF:
4546                                            if (!dev_name_match(&rp6->
4547                                                ipv6RouteIfIndex, fp->u.f_ifname))
4548                                                    /* 'for' loop 2 */
4549                                                    continue;
4550                                            break;
4551                                    case FK_DST:
```



```
4552                                            if (!v6_addr_match(&rp6->ipv6RouteDest,
4553                                                rp6->ipv6RoutePfxLength, fp))
4554                                                    /* 'for' loop 2 */
4555                                                    continue;
4556                                            break;
4557                                    case FK_FLAGS:
4558                                            if ((flag_b & fp->u.f.f_flagset) !=
4559                                                fp->u.f.f_flagset ||
4560                                                (flag_b & fp->u.f.f_flagclear))
4561                                                    /* 'for' loop 2 */
4562                                                    continue;
4563                                            break;
4564                                    }
4565                                    break;
4566                            } /* 'for' loop 2 ends */
4567                            if (fp == NULL)
4568                                    return (B_FALSE);
4569                    }
4570            /* 'for' loop 1 ends */
4571            return (B_TRUE);
4572    }

4574    /*
4575     * Given an IPv6 MIB2 route entry, form the list of flags for the
4576     * route.
4577     */
4578    static uint_t
4579    form_v6_route_flags(const mib2_ipv6RouteEntry_t *rp6, char *flags)
4580    {
4581            uint_t flag_b;

4583            flag_b = FLF_U;
4584            (void) strcpy(flags, "U");
4585            /* RTF_INDIRECT wins over RTF_GATEWAY - don't display both */
4586            if (rp6->ipv6RouteInfo.re_flags & RTF_INDIRECT) {
4587                    (void) strcat(flags, "I");
4588                    flag_b |= FLF_I;
4589            } else if (rp6->ipv6RouteInfo.re_ire_type & IRE_OFFLINK) {
4590                    (void) strcat(flags, "G");
4591                    flag_b |= FLF_G;
4592            }

4594            /* IRE_IF_CLONE wins over RTF_HOST - don't display both */
4595            if (rp6->ipv6RouteInfo.re_ire_type & IRE_IF_CLONE) {
4596                    (void) strcat(flags, "C");
4597                    flag_b |= FLF_C;
4598            } else if (rp6->ipv6RoutePfxLength == IPV6_ABITS) {
4599                    (void) strcat(flags, "H");
4600                    flag_b |= FLF_H;
4601            }

4603            if (rp6->ipv6RouteInfo.re_flags & RTF_DYNAMIC) {
4604                    (void) strcat(flags, "D");
4605                    flag_b |= FLF_D;
4606            }
4607            if (rp6->ipv6RouteInfo.re_ire_type == IRE_LOCAL) {      /* Local */
4608                    (void) strcat(flags, "L");
4609                    flag_b |= FLF_L;
4610            }
4611            if (rp6->ipv6RouteInfo.re_flags & RTF_MULTIRT) {
4612                    (void) strcat(flags, "M");                      /* Multiroute */
4613                    flag_b |= FLF_M;
4614            }
4615            if (rp6->ipv6RouteInfo.re_flags & RTF_SETSRC) {
4616                    (void) strcat(flags, "S");                      /* Setsrc */
4617                    flag_b |= FLF_S;
```

```
4618         }
4619         if (rp6->ipv6RouteInfo.re_flags & RTF_REJECT) {
4620                 (void) strcat(flags, "R");
4621                 flag_b |= FLF_R;
4622         }
4623         if (rp6->ipv6RouteInfo.re_flags & RTF_BLACKHOLE) {
4624                 (void) strcat(flags, "B");
4625                 flag_b |= FLF_B;
4626         }
4627         if (rp6->ipv6RouteInfo.re_flags & RTF_ZONE) {
4628                 (void) strcat(flags, "Z");
4629                 flag_b |= FLF_Z;
4630         }
4631         return (flag_b);
4632 }

4634 static const char ire_hdr_v6[] =
4635 "\n%s Table: IPv6\n";
4636 static const char ire_hdr_v6_verbose[] =
4637 "  Destination/Mask            Gateway                     If    MTU  "
4638 "Ref Flags  Out   In/Fwd %s\n"
4639 "--------------------------- --------------------------- ----- ----- "
4640 "--- ----- ------ ------ %s\n";
4641 static const char ire_hdr_v6_normal[] =
4642 "  Destination/Mask            Gateway                   Flags Ref   Use  "
4643 "  If   %s\n"
4644 "--------------------------- --------------------------- ----- --- ------- "
4645 "----- %s\n";

4647 static boolean_t
4648 ire_report_item_v6(const mib2_ipv6RouteEntry_t *rp6, boolean_t first,
4649     const sec_attr_list_t *attrs)
4650 {
4651         char                    dstbuf[MAXHOSTNAMELEN + 1];
4652         char                    gwbuf[MAXHOSTNAMELEN + 1];
4653         char                    ifname[LIFNAMSIZ + 1];
4654         char                    flags[10];      /* RTF_ flags */
4655         uint_t                  flag_b;

4657         if (!(Aflag || (rp6->ipv6RouteInfo.re_ire_type != IRE_IF_CLONE &&
4658             rp6->ipv6RouteInfo.re_ire_type != IRE_MULTICAST &&
4659             rp6->ipv6RouteInfo.re_ire_type != IRE_NOROUTE &&
4660             rp6->ipv6RouteInfo.re_ire_type != IRE_LOCAL))) {
4661                 return (first);
4662         }

4664         flag_b = form_v6_route_flags(rp6, flags);

4666         if (!ire_filter_match_v6(rp6, flag_b))
4667                 return (first);

4669         if (first) {
4670                 (void) printf(ire_hdr_v6, Vflag ? "IRE" : "Routing");
4671                 (void) printf(Vflag ? ire_hdr_v6_verbose : ire_hdr_v6_normal,
4672                     RSECflag ? "  Gateway security attributes  " : "",
4673                     RSECflag ? "-------------------------------" : "");
4674                 first = B_FALSE;
4675         }

4677         if (Vflag) {
4678                 (void) printf("%-27s %-27s %-5s %5u %3u "
4679                     "%-5s %6u %6u %s\n",
4680                     pr_prefix6(&rp6->ipv6RouteDest,
4681                     rp6->ipv6RoutePfxLength, dstbuf, sizeof (dstbuf)),
4682                     IN6_IS_ADDR_UNSPECIFIED(&rp6->ipv6RouteNextHop) ?
4683                     "    --" :
```



```
4684                     pr_addr6(&rp6->ipv6RouteNextHop, gwbuf, sizeof (gwbuf)),
4685                     octetstr(&rp6->ipv6RouteIfIndex, 'a',
4686                     ifname, sizeof (ifname)),
4687                     rp6->ipv6RouteInfo.re_max_frag,
4688                     rp6->ipv6RouteInfo.re_ref,
4689                     flags,
4690                     rp6->ipv6RouteInfo.re_obpkt,
4691                     rp6->ipv6RouteInfo.re_ibpkt,
4692                     pr_secattr(attrs));
4693         } else {
4694                 (void) printf("%-27s %-27s %-5s %3u %7u %-5s %s\n",
4695                     pr_prefix6(&rp6->ipv6RouteDest,
4696                     rp6->ipv6RoutePfxLength, dstbuf, sizeof (dstbuf)),
4697                     IN6_IS_ADDR_UNSPECIFIED(&rp6->ipv6RouteNextHop) ?
4698                     "    --" :
4699                     pr_addr6(&rp6->ipv6RouteNextHop, gwbuf, sizeof (gwbuf)),
4700                     flags,
4701                     rp6->ipv6RouteInfo.re_ref,
4702                     rp6->ipv6RouteInfo.re_obpkt + rp6->ipv6RouteInfo.re_ibpkt,
4703                     octetstr(&rp6->ipv6RouteIfIndex, 'a',
4704                     ifname, sizeof (ifname)),
4705                     pr_secattr(attrs));
4706         }
4707         return (first);
4708 }

4710 /*
4711  * Common attribute-gathering routine for all transports.
4712  */
4713 static mib2_transportMLPEntry_t **
4714 gather_attrs(const mib_item_t *item, int group, int mib_id, int esize)
4715 {
4716         int transport_count = 0;
4717         const mib_item_t *iptr;
4718         mib2_transportMLPEntry_t **attrs, *tme;

4720         for (iptr = item; iptr != NULL; iptr = iptr->next_item) {
4721                 if (iptr->group == group && iptr->mib_id == mib_id)
4722                         transport_count += iptr->length / esize;
4723         }
4724         if (transport_count <= 0)
4725                 return (NULL);
4726         attrs = calloc(transport_count, sizeof (*attrs));
4727         if (attrs == NULL) {
4728                 perror("gather_attrs calloc failed");
4729                 return (NULL);
4730         }
4731         for (iptr = item; iptr != NULL; iptr = iptr->next_item) {
4732                 if (iptr->group == group && iptr->mib_id == EXPER_XPORT_MLP) {
4733                         for (tme = iptr->valp;
4734                             (char *)tme < (char *)iptr->valp + iptr->length;
4735                             /* LINTED: (note 1) */
4736                             tme = (mib2_transportMLPEntry_t *)((char *)tme +
4737                             transportMLPSize)) {
4738                                 attrs[tme->tme_connidx] = tme;
4739                         }
4740                 }
4741         }
4742         return (attrs);
4743 }

4745 static void
4746 print_transport_label(const mib2_transportMLPEntry_t *attr)
4747 {
4748         if (!RSECflag || attr == NULL ||
4749             !(attr->tme_flags & MIB2_TMEF_IS_LABELED))
```

```
4750                 return;

4752         if (bisinvalid(&attr->tme_label)) {
4753                 (void) printf("   INVALID\n");
4754         } else if (!blequal(&attr->tme_label, zone_security_label)) {
4755                 char *sl_str;

4757                 sl_str = sl_to_str(&attr->tme_label);
4758                 (void) printf("   %s\n", sl_str);
4759                 free(sl_str);
4760         }
4761 }

4763 /* ----------------------------- TCP_REPORT------------------------------ */

4765 static const char tcp_hdr_v4[] =
4766 "\nTCP: IPv4\n";
4767 static const char tcp_hdr_v4_compat[] =
4768 "\nTCP\n";
4769 static const char tcp_hdr_v4_verbose[] =
4770 "Local/Remote Address Swind  Snext     Suna   Rwind  Rnext     Rack   "
4771 " Rto   Mss     State\n"
4772 "-------------------- ----- -------- -------- ----- -------- -------- "
4773 "----- ----- -----------\n";
4774 static const char tcp_hdr_v4_normal[] =
4775 "   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q "
4776 "   State\n"
4777 "-------------------- -------------------- ----- ------ ----- ------ "
4778 "-----------\n";

4780 static const char tcp_hdr_v6[] =
4781 "\nTCP: IPv6\n";
4782 static const char tcp_hdr_v6_verbose[] =
4783 "Local/Remote Address               Swind  Snext     Suna   Rwind  Rnext   "
4784 "  Rack    Rto   Mss    State      If\n"
4785 "-------------------------------- ----- -------- -------- ----- -------- "
4786 "-------- ----- ----- ----------- -----\n";
4787 static const char tcp_hdr_v6_normal[] =
4788 "   Local Address                       Remote Address                "
4789 "Swind Send-Q Rwind Recv-Q   State      If\n"
4790 "--------------------------------- ---------------------------------- "
4791 "----- ------ ----- ------ ----------- -----\n";

4793 static boolean_t tcp_report_item_v4(const mib2_tcpConnEntry_t *,
4794     boolean_t first, const mib2_transportMLPEntry_t *);
4795 static boolean_t tcp_report_item_v6(const mib2_tcp6ConnEntry_t *,
4796     boolean_t first, const mib2_transportMLPEntry_t *);

4798 static void
4799 tcp_report(const mib_item_t *item)
4800 {
4801         int                     jtemp = 0;
4802         boolean_t               print_hdr_once_v4 = B_TRUE;
4803         boolean_t               print_hdr_once_v6 = B_TRUE;
4804         mib2_tcpConnEntry_t     *tp;
4805         mib2_tcp6ConnEntry_t    *tp6;
4806         mib2_transportMLPEntry_t **v4_attrs, **v6_attrs;
4807         mib2_transportMLPEntry_t **v4a, **v6a;
4808         mib2_transportMLPEntry_t *aptr;

4810         if (!protocol_selected(IPPROTO_TCP))
4811                 return;

4813         /*
4814          * Preparation pass: the kernel returns separate entries for TCP
4815          * connection table entries and Multilevel Port attributes.  We loop
```



```
4816          * through the attributes first and set up an array for each address
4817          * family.
4818          */
4819         v4_attrs = family_selected(AF_INET) && RSECflag ?
4820             gather_attrs(item, MIB2_TCP, MIB2_TCP_CONN, tcpConnEntrySize) :
4821             NULL;
4822         v6_attrs = family_selected(AF_INET6) && RSECflag ?
4823             gather_attrs(item, MIB2_TCP6, MIB2_TCP6_CONN, tcp6ConnEntrySize) :
4824             NULL;

4826         /* 'for' loop 1: */
4827         v4a = v4_attrs;
4828         v6a = v6_attrs;
4829         for (; item != NULL; item = item->next_item) {
4830                 if (Xflag) {
4831                         (void) printf("\n--- Entry %d ---\n", ++jtemp);
4832                         (void) printf("Group = %d, mib_id = %d, "
4833                             "length = %d, valp = 0x%p\n",
4834                             item->group, item->mib_id,
4835                             item->length, item->valp);
4836                 }

4838                 if (!((item->group == MIB2_TCP &&
4839                     item->mib_id == MIB2_TCP_CONN) ||
4840                     (item->group == MIB2_TCP6 &&
4841                     item->mib_id == MIB2_TCP6_CONN)))
4842                         continue; /* 'for' loop 1 */

4844                 if (item->group == MIB2_TCP && !family_selected(AF_INET))
4845                         continue; /* 'for' loop 1 */
4846                 else if (item->group == MIB2_TCP6 && !family_selected(AF_INET6))
4847                         continue; /* 'for' loop 1 */

4849                 if (item->group == MIB2_TCP) {
4850                         for (tp = (mib2_tcpConnEntry_t *)item->valp;
4851                             (char *)tp < (char *)item->valp + item->length;
4852                             /* LINTED: (note 1) */
4853                             tp = (mib2_tcpConnEntry_t *)((char *)tp +
4854                             tcpConnEntrySize)) {
4855                                 aptr = v4a == NULL ? NULL : *v4a++;
4856                                 print_hdr_once_v4 = tcp_report_item_v4(tp,
4857                                     print_hdr_once_v4, aptr);
4858                         }
4859                 } else {
4860                         for (tp6 = (mib2_tcp6ConnEntry_t *)item->valp;
4861                             (char *)tp6 < (char *)item->valp + item->length;
4862                             /* LINTED: (note 1) */
4863                             tp6 = (mib2_tcp6ConnEntry_t *)((char *)tp6 +
4864                             tcp6ConnEntrySize)) {
4865                                 aptr = v6a == NULL ? NULL : *v6a++;
4866                                 print_hdr_once_v6 = tcp_report_item_v6(tp6,
4867                                     print_hdr_once_v6, aptr);
4868                         }
4869                 }
4870         } /* 'for' loop 1 ends */
4871         (void) fflush(stdout);

4873         if (v4_attrs != NULL)
4874                 free(v4_attrs);
4875         if (v6_attrs != NULL)
4876                 free(v6_attrs);
4877 }

4879 static boolean_t
4880 tcp_report_item_v4(const mib2_tcpConnEntry_t *tp, boolean_t first,
4881     const mib2_transportMLPEntry_t *attr)
```

```
4882 {
4883         /*
4884          * lname and fname below are for the hostname as well as the portname
4885          * There is no limit on portname length so we assume MAXHOSTNAMELEN
4886          * as the limit
4887          */
4888         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4889         char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];

4891         if (!(Aflag || tp->tcpConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
4892                 return (first); /* Nothing to print */

4894         if (first) {
4895                 (void) printf(v4compat ? tcp_hdr_v4_compat : tcp_hdr_v4);
4896                 (void) printf(Vflag ? tcp_hdr_v4_verbose : tcp_hdr_v4_normal);
4897         }

4899         if (Vflag) {
4900                 (void) printf("%-20s\n%-20s %5u %08x %08x %5u %08x %08x "
4901                     "%5u %5u %s\n",
4902                     pr_ap(tp->tcpConnLocalAddress,
4903                     tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4904                     pr_ap(tp->tcpConnRemAddress,
4905                     tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4906                     tp->tcpConnEntryInfo.ce_swnd,
4907                     tp->tcpConnEntryInfo.ce_snxt,
4908                     tp->tcpConnEntryInfo.ce_suna,
4909                     tp->tcpConnEntryInfo.ce_rwnd,
4910                     tp->tcpConnEntryInfo.ce_rnxt,
4911                     tp->tcpConnEntryInfo.ce_rack,
4912                     tp->tcpConnEntryInfo.ce_rto,
4913                     tp->tcpConnEntryInfo.ce_mss,
4914                     mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4915         } else {
4916                 int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
4917                     (int)tp->tcpConnEntryInfo.ce_suna - 1;
4918                 int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
4919                     (int)tp->tcpConnEntryInfo.ce_rack;

4921                 (void) printf("%-20s %-20s %5u %6d %5u %6d %s\n",
4922                     pr_ap(tp->tcpConnLocalAddress,
4923                     tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4924                     pr_ap(tp->tcpConnRemAddress,
4925                     tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4926                     tp->tcpConnEntryInfo.ce_swnd,
4927                     (sq >= 0) ? sq : 0,
4928                     tp->tcpConnEntryInfo.ce_rwnd,
4929                     (rq >= 0) ? rq : 0,
4930                     mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4931         }

4933         print_transport_label(attr);

4935         return (B_FALSE);
4936 }

4938 static boolean_t
4939 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6, boolean_t first,
4940     const mib2_transportMLPEntry_t *attr)
4941 {
4942         /*
4943          * lname and fname below are for the hostname as well as the portname
4944          * There is no limit on portname length so we assume MAXHOSTNAMELEN
4945          * as the limit
4946          */
4947         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
```



```
4948         char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4949         char    ifname[LIFNAMSIZ + 1];
4950         char    *ifnamep;

4952         if (!(Aflag || tp6->tcp6ConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
4953                 return (first); /* Nothing to print */

4955         if (first) {
4956                 (void) printf(tcp_hdr_v6);
4957                 (void) printf(Vflag ? tcp_hdr_v6_verbose : tcp_hdr_v6_normal);
4958         }

4960         ifnamep = (tp6->tcp6ConnIfIndex != 0) ?
4961             if_indextoname(tp6->tcp6ConnIfIndex, ifname) : NULL;
4962         if (ifnamep == NULL)
4963                 ifnamep = "";

4965         if (Vflag) {
4966                 (void) printf("%-33s\n%-33s %5u %08x %08x %5u %08x %08x "
4967                     "%5u %5u %-11s %s\n",
4968                     pr_ap6(&tp6->tcp6ConnLocalAddress,
4969                     tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
4970                     pr_ap6(&tp6->tcp6ConnRemAddress,
4971                     tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
4972                     tp6->tcp6ConnEntryInfo.ce_swnd,
4973                     tp6->tcp6ConnEntryInfo.ce_snxt,
4974                     tp6->tcp6ConnEntryInfo.ce_suna,
4975                     tp6->tcp6ConnEntryInfo.ce_rwnd,
4976                     tp6->tcp6ConnEntryInfo.ce_rnxt,
4977                     tp6->tcp6ConnEntryInfo.ce_rack,
4978                     tp6->tcp6ConnEntryInfo.ce_rto,
4979                     tp6->tcp6ConnEntryInfo.ce_mss,
4980                     mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
4981                     ifnamep);
4982         } else {
4983                 int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
4984                     (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
4985                 int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
4986                     (int)tp6->tcp6ConnEntryInfo.ce_rack;

4988                 (void) printf("%-33s %-33s %5u %6d %5u %6d %-11s %s\n",
4989                     pr_ap6(&tp6->tcp6ConnLocalAddress,
4990                     tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
4991                     pr_ap6(&tp6->tcp6ConnRemAddress,
4992                     tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
4993                     tp6->tcp6ConnEntryInfo.ce_swnd,
4994                     (sq >= 0) ? sq : 0,
4995                     tp6->tcp6ConnEntryInfo.ce_rwnd,
4996                     (rq >= 0) ? rq : 0,
4997                     mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
4998                     ifnamep);
4999         }

5001         print_transport_label(attr);

5003         return (B_FALSE);
5004 }

5006 /* ------------------------------ UDP_REPORT------------------------------ */

5008 static boolean_t udp_report_item_v4(const mib2_udpEntry_t *ude,
5009     boolean_t first, const mib2_transportMLPEntry_t *attr);
5010 static boolean_t udp_report_item_v6(const mib2_udp6Entry_t *ude6,
5011     boolean_t first, const mib2_transportMLPEntry_t *attr);

5013 static const char udp_hdr_v4[] =
```

```
"   Local Address        Remote Address      State\n"
"-------------------- -------------------- ----------\n";

static const char udp_hdr_v6[] =
"   Local Address                     Remote Address                "
"  State      If\n"
"------------------------------ ------------------------------ "
"---------- -----\n";

static void
udp_report(const mib_item_t *item)
{
        int                     jtemp = 0;
        boolean_t               print_hdr_once_v4 = B_TRUE;
        boolean_t               print_hdr_once_v6 = B_TRUE;
        mib2_udpEntry_t         *ude;
        mib2_udp6Entry_t        *ude6;
        mib2_transportMLPEntry_t **v4_attrs, **v6_attrs;
        mib2_transportMLPEntry_t **v4a, **v6a;
        mib2_transportMLPEntry_t *aptr;

        if (!protocol_selected(IPPROTO_UDP))
                return;

        /*
         * Preparation pass: the kernel returns separate entries for UDP
         * connection table entries and Multilevel Port attributes.  We loop
         * through the attributes first and set up an array for each address
         * family.
         */
        v4_attrs = family_selected(AF_INET) && RSECflag ?
            gather_attrs(item, MIB2_UDP, MIB2_UDP_ENTRY, udpEntrySize) : NULL;
        v6_attrs = family_selected(AF_INET6) && RSECflag ?
            gather_attrs(item, MIB2_UDP6, MIB2_UDP6_ENTRY, udp6EntrySize) :
            NULL;

        v4a = v4_attrs;
        v6a = v6_attrs;
        /* 'for' loop 1: */
        for (; item; item = item->next_item) {
                if (Xflag) {
                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
                        (void) printf("Group = %d, mib_id = %d, "
                            "length = %d, valp = 0x%p\n",
                            item->group, item->mib_id,
                            item->length, item->valp);
                }
                if (!((item->group == MIB2_UDP &&
                    item->mib_id == MIB2_UDP_ENTRY) ||
                    (item->group == MIB2_UDP6 &&
                    item->mib_id == MIB2_UDP6_ENTRY)))
                        continue; /* 'for' loop 1 */

                if (item->group == MIB2_UDP && !family_selected(AF_INET))
                        continue; /* 'for' loop 1 */
                else if (item->group == MIB2_UDP6 && !family_selected(AF_INET6))
                        continue; /* 'for' loop 1 */

                /*      xxx.xxx.xxx.xxx,pppp  sss... */
                if (item->group == MIB2_UDP) {
                        for (ude = (mib2_udpEntry_t *)item->valp;
                            (char *)ude < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            ude = (mib2_udpEntry_t *)((char *)ude +
                            udpEntrySize)) {
                                aptr = v4a == NULL ? NULL : *v4a++;
```



```
                                print_hdr_once_v4 = udp_report_item_v4(ude,
                                    print_hdr_once_v4, aptr);
                        }
                } else {
                        for (ude6 = (mib2_udp6Entry_t *)item->valp;
                            (char *)ude6 < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            ude6 = (mib2_udp6Entry_t *)((char *)ude6 +
                            udp6EntrySize)) {
                                aptr = v6a == NULL ? NULL : *v6a++;
                                print_hdr_once_v6 = udp_report_item_v6(ude6,
                                    print_hdr_once_v6, aptr);
                        }
                }
        } /* 'for' loop 1 ends */
        (void) fflush(stdout);

        if (v4_attrs != NULL)
                free(v4_attrs);
        if (v6_attrs != NULL)
                free(v6_attrs);
}

static boolean_t
udp_report_item_v4(const mib2_udpEntry_t *ude, boolean_t first,
    const mib2_transportMLPEntry_t *attr)
{
        char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
                        /* hostname + portname */

        if (!(Aflag || ude->udpEntryInfo.ue_state >= MIB2_UDP_connected))
                return (first); /* Nothing to print */

        if (first) {
                (void) printf(v4compat ? "\nUDP\n" : "\nUDP: IPv4\n");
                (void) printf(udp_hdr_v4);
                first = B_FALSE;
        }

        (void) printf("%-20s ",
            pr_ap(ude->udpLocalAddress, ude->udpLocalPort, "udp",
            lname, sizeof (lname)));
        (void) printf("%-20s %s\n",
            ude->udpEntryInfo.ue_state == MIB2_UDP_connected ?
            pr_ap(ude->udpEntryInfo.ue_RemoteAddress,
            ude->udpEntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
            "",
            miudp_state(ude->udpEntryInfo.ue_state, attr));

        print_transport_label(attr);

        return (first);
}

static boolean_t
udp_report_item_v6(const mib2_udp6Entry_t *ude6, boolean_t first,
    const mib2_transportMLPEntry_t *attr)
{
        char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
                        /* hostname + portname */
        char    ifname[LIFNAMSIZ + 1];
        const char *ifnamep;

        if (!(Aflag || ude6->udp6EntryInfo.ue_state >= MIB2_UDP_connected))
                return (first); /* Nothing to print */
```

```
5146         if (first) {
5147                 (void) printf("\nUDP: IPv6\n");
5148                 (void) printf(udp_hdr_v6);
5149                 first = B_FALSE;
5150         }

5152         ifnamep = (ude6->udp6IfIndex != 0) ?
5153             if_indextoname(ude6->udp6IfIndex, ifname) : NULL;

5155         (void) printf("%-33s ",
5156             pr_ap6(&ude6->udp6LocalAddress,
5157             ude6->udp6LocalPort, "udp", lname, sizeof (lname)));
5158         (void) printf("%-33s %-10s %s\n",
5159             ude6->udp6EntryInfo.ue_state == MIB2_UDP_connected ?
5160             pr_ap6(&ude6->udp6EntryInfo.ue_RemoteAddress,
5161             ude6->udp6EntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5162             "",
5163             miudp_state(ude6->udp6EntryInfo.ue_state, attr),
5164             ifnamep == NULL ? "" : ifnamep);

5166         print_transport_label(attr);

5168         return (first);
5169 }

5171 /* ---------------------------- SCTP_REPORT----------------------------- */

5173 static const char sctp_hdr[] =
5174 "\nSCTP:";
5175 static const char sctp_hdr_normal[] =
5176 "        Local Address                 Remote Address          "
5177 "Swind  Send-Q Rwind  Recv-Q StrsI/O  State\n"
5178 "------------------------------ ------------------------------ "
5179 "------ ------ ------ ------ ------- -----------";

5181 static const char *
5182 nssctp_state(int state, const mib2_transportMLPEntry_t *attr)
5183 {
5184         static char sctpsbuf[50];
5185         const char *cp;

5187         switch (state) {
5188         case MIB2_SCTP_closed:
5189                 cp = "CLOSED";
5190                 break;
5191         case MIB2_SCTP_cookieWait:
5192                 cp = "COOKIE_WAIT";
5193                 break;
5194         case MIB2_SCTP_cookieEchoed:
5195                 cp = "COOKIE_ECHOED";
5196                 break;
5197         case MIB2_SCTP_established:
5198                 cp = "ESTABLISHED";
5199                 break;
5200         case MIB2_SCTP_shutdownPending:
5201                 cp = "SHUTDOWN_PENDING";
5202                 break;
5203         case MIB2_SCTP_shutdownSent:
5204                 cp = "SHUTDOWN_SENT";
5205                 break;
5206         case MIB2_SCTP_shutdownReceived:
5207                 cp = "SHUTDOWN_RECEIVED";
5208                 break;
5209         case MIB2_SCTP_shutdownAckSent:
5210                 cp = "SHUTDOWN_ACK_SENT";
5211                 break;
```



```
5212         case MIB2_SCTP_listen:
5213                 cp = "LISTEN";
5214                 break;
5215         default:
5216                 (void) snprintf(sctpsbuf, sizeof (sctpsbuf),
5217                     "UNKNOWN STATE(%d)", state);
5218                 cp = sctpsbuf;
5219                 break;
5220         }

5222         if (RSECflag && attr != NULL && attr->tme_flags != 0) {
5223                 if (cp != sctpsbuf) {
5224                         (void) strlcpy(sctpsbuf, cp, sizeof (sctpsbuf));
5225                         cp = sctpsbuf;
5226                 }
5227                 if (attr->tme_flags & MIB2_TMEF_PRIVATE)
5228                         (void) strlcat(sctpsbuf, " P", sizeof (sctpsbuf));
5229                 if (attr->tme_flags & MIB2_TMEF_SHARED)
5230                         (void) strlcat(sctpsbuf, " S", sizeof (sctpsbuf));
5231         }

5233         return (cp);
5234 }

5236 static const mib2_sctpConnRemoteEntry_t *
5237 sctp_getnext_rem(const mib_item_t **itemp,
5238     const mib2_sctpConnRemoteEntry_t *current, uint32_t associd)
5239 {
5240         const mib_item_t *item = *itemp;
5241         const mib2_sctpConnRemoteEntry_t        *sre;

5243         for (; item != NULL; item = item->next_item, current = NULL) {
5244                 if (!(item->group == MIB2_SCTP &&
5245                     item->mib_id == MIB2_SCTP_CONN_REMOTE)) {
5246                         continue;
5247                 }

5249                 if (current != NULL) {
5250                         /* LINTED: (note 1) */
5251                         sre = (const mib2_sctpConnRemoteEntry_t *)
5252                             ((const char *)current + sctpRemoteEntrySize);
5253                 } else {
5254                         sre = item->valp;
5255                 }
5256                 for (; (char *)sre < (char *)item->valp + item->length;
5257                     /* LINTED: (note 1) */
5258                     sre = (const mib2_sctpConnRemoteEntry_t *)
5259                     ((const char *)sre + sctpRemoteEntrySize)) {
5260                         if (sre->sctpAssocId != associd) {
5261                                 continue;
5262                         }
5263                         *itemp = item;
5264                         return (sre);
5265                 }
5266         }
5267         *itemp = NULL;
5268         return (NULL);
5269 }

5271 static const mib2_sctpConnLocalEntry_t *
5272 sctp_getnext_local(const mib_item_t **itemp,
5273     const mib2_sctpConnLocalEntry_t *current, uint32_t associd)
5274 {
5275         const mib_item_t *item = *itemp;
5276         const mib2_sctpConnLocalEntry_t *sle;
```

```
5278         for (; item != NULL; item = item->next_item, current = NULL) {
5279                 if (!(item->group == MIB2_SCTP &&
5280                     item->mib_id == MIB2_SCTP_CONN_LOCAL)) {
5281                         continue;
5282                 }
5283
5284                 if (current != NULL) {
5285                         /* LINTED: (note 1) */
5286                         sle = (const mib2_sctpConnLocalEntry_t *)
5287                             ((const char *)current + sctpLocalEntrySize);
5288                 } else {
5289                         sle = item->valp;
5290                 }
5291                 for (; (char *)sle < (char *)item->valp + item->length;
5292                     /* LINTED: (note 1) */
5293                     sle = (const mib2_sctpConnLocalEntry_t *)
5294                     ((const char *)sle + sctpLocalEntrySize)) {
5295                         if (sle->sctpAssocId != associd) {
5296                                 continue;
5297                         }
5298                         *itemp = item;
5299                         return (sle);
5300                 }
5301         }
5302         *itemp = NULL;
5303         return (NULL);
5304 }
5305
5306 static void
5307 sctp_pr_addr(int type, char *name, int namelen, const in6_addr_t *addr,
5308     int port)
5309 {
5310         ipaddr_t        v4addr;
5311         in6_addr_t      v6addr;
5312
5313         /*
5314          * Address is either a v4 mapped or v6 addr. If
5315          * it's a v4 mapped, convert to v4 before
5316          * displaying.
5317          */
5318         switch (type) {
5319         case MIB2_SCTP_ADDR_V4:
5320                 /* v4 */
5321                 v6addr = *addr;
5322
5323                 IN6_V4MAPPED_TO_IPADDR(&v6addr, v4addr);
5324                 if (port > 0) {
5325                         (void) pr_ap(v4addr, port, "sctp", name, namelen);
5326                 } else {
5327                         (void) pr_addr(v4addr, name, namelen);
5328                 }
5329                 break;
5330
5331         case MIB2_SCTP_ADDR_V6:
5332                 /* v6 */
5333                 if (port > 0) {
5334                         (void) pr_ap6(addr, port, "sctp", name, namelen);
5335                 } else {
5336                         (void) pr_addr6(addr, name, namelen);
5337                 }
5338                 break;
5339
5340         default:
5341                 (void) snprintf(name, namelen, "<unknown addr type>");
5342                 break;
5343         }
```



```
5344 }
5345
5346 static void
5347 sctp_conn_report_item(const mib_item_t *head, const mib2_sctpConnEntry_t *sp,
5348     const mib2_transportMLPEntry_t *attr)
5349 {
5350         char            lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5351         char            fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5352         const mib2_sctpConnRemoteEntry_t        *sre = NULL;
5353         const mib2_sctpConnLocalEntry_t *sle = NULL;
5354         const mib_item_t *local = head;
5355         const mib_item_t *remote = head;
5356         uint32_t        id = sp->sctpAssocId;
5357         boolean_t       printfirst = B_TRUE;
5358
5359         sctp_pr_addr(sp->sctpAssocRemPrimAddrType, fname, sizeof (fname),
5360             &sp->sctpAssocRemPrimAddr, sp->sctpAssocRemPort);
5361         sctp_pr_addr(sp->sctpAssocRemPrimAddrType, lname, sizeof (lname),
5362             &sp->sctpAssocLocPrimAddr, sp->sctpAssocLocalPort);
5363
5364         (void) printf("%-31s %-31s %6u %6d %6u %6d %3d/%-3d %s\n",
5365             lname, fname,
5366             sp->sctpConnEntryInfo.ce_swnd,
5367             sp->sctpConnEntryInfo.ce_sendq,
5368             sp->sctpConnEntryInfo.ce_rwnd,
5369             sp->sctpConnEntryInfo.ce_recvq,
5370             sp->sctpAssocInStreams, sp->sctpAssocOutStreams,
5371             nssctp_state(sp->sctpAssocState, attr));
5372
5373         print_transport_label(attr);
5374
5375         if (!Vflag) {
5376                 return;
5377         }
5378
5379         /* Print remote addresses/local addresses on following lines */
5380         while ((sre = sctp_getnext_rem(&remote, sre, id)) != NULL) {
5381                 if (!IN6_ARE_ADDR_EQUAL(&sre->sctpAssocRemAddr,
5382                     &sp->sctpAssocRemPrimAddr)) {
5383                         if (printfirst == B_TRUE) {
5384                                 (void) fputs("\t<Remote: ", stdout);
5385                                 printfirst = B_FALSE;
5386                         } else {
5387                                 (void) fputs(", ", stdout);
5388                         }
5389                         sctp_pr_addr(sre->sctpAssocRemAddrType, fname,
5390                             sizeof (fname), &sre->sctpAssocRemAddr, -1);
5391                         if (sre->sctpAssocRemAddrActive == MIB2_SCTP_ACTIVE) {
5392                                 (void) fputs(fname, stdout);
5393                         } else {
5394                                 (void) printf("(%s)", fname);
5395                         }
5396                 }
5397         }
5398         if (printfirst == B_FALSE) {
5399                 (void) puts(">");
5400                 printfirst = B_TRUE;
5401         }
5402         while ((sle = sctp_getnext_local(&local, sle, id)) != NULL) {
5403                 if (!IN6_ARE_ADDR_EQUAL(&sle->sctpAssocLocalAddr,
5404                     &sp->sctpAssocLocPrimAddr)) {
5405                         if (printfirst == B_TRUE) {
5406                                 (void) fputs("\t<Local: ", stdout);
5407                                 printfirst = B_FALSE;
5408                         } else {
5409                                 (void) fputs(", ", stdout);
```

```
                        }
                        sctp_pr_addr(sle->sctpAssocLocalAddrType, lname,
                            sizeof (lname), &sle->sctpAssocLocalAddr, -1);
                        (void) fputs(lname, stdout);
                }
        }
        if (printfirst == B_FALSE) {
                (void) puts(">");
        }
}

static void
sctp_report(const mib_item_t *item)
{
        const mib_item_t               *head;
        const mib2_sctpConnEntry_t      *sp;
        boolean_t               first = B_TRUE;
        mib2_transportMLPEntry_t **attrs, **aptr;
        mib2_transportMLPEntry_t *attr;

        /*
         * Preparation pass: the kernel returns separate entries for SCTP
         * connection table entries and Multilevel Port attributes.  We loop
         * through the attributes first and set up an array for each address
         * family.
         */
        attrs = RSECflag ?
            gather_attrs(item, MIB2_SCTP, MIB2_SCTP_CONN, sctpEntrySize) :
            NULL;

        aptr = attrs;
        head = item;
        for (; item != NULL; item = item->next_item) {

                if (!(item->group == MIB2_SCTP &&
                    item->mib_id == MIB2_SCTP_CONN))
                        continue;

                for (sp = item->valp;
                    (char *)sp < (char *)item->valp + item->length;
                    /* LINTED: (note 1) */
                    sp = (mib2_sctpConnEntry_t *)((char *)sp + sctpEntrySize)) {
                        attr = aptr == NULL ? NULL : *aptr++;
                        if (Aflag ||
                            sp->sctpAssocState >= MIB2_SCTP_established) {
                                if (first == B_TRUE) {
                                        (void) puts(sctp_hdr);
                                        (void) puts(sctp_hdr_normal);
                                        first = B_FALSE;
                                }
                                sctp_conn_report_item(head, sp, attr);
                        }
                }
        }
        if (attrs != NULL)
                free(attrs);
}

/* ---------------------------- DCCP_REPORT------------------------------ */

static const char dccp_hdr_v4[] =
"\nDCCP: IPv4\n";
static const char dccp_hdr_v4_compat[] =
"\nDCCP\n";
static const char dccp_hdr_v4_verbose[] =
"Local/Remote Address Swind  Snext     Suna   Rwind  Rnext     Rack   "
```



```
" Rto   Mss     State\n"
"-------------------- ----- -------- -------- ----- -------- -------- "
"----- ----- ----------\n";
static const char dccp_hdr_v4_normal[] =
"   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q "
"   State\n"
"-------------------- -------------------- ----- ------ ----- ------ "
"----------\n";

static const char dccp_hdr_v6[] =
"\nDCCP: IPv6\n";
static const char dccp_hdr_v6_verbose[] =
"Local/Remote Address                Swind  Snext     Suna   Rwind  Rnext   "
"  Rack    Rto   Mss    State      If\n"
"-------------------------------- ----- -------- -------- ----- -------- "
"-------- ----- ----- ---------- -----\n";
static const char dccp_hdr_v6_normal[] =
"   Local Address                         Remote Address                "
"Swind Send-Q Rwind Recv-Q   State      If\n"
"-------------------------------- -------------------------------- "
"----- ------ ----- ------ ---------- -----\n";

static boolean_t dccp_report_item_v4(const mib2_dccpConnEntry_t *,
    boolean_t, const mib2_transportMLPEntry_t *);
static boolean_t dccp_report_item_v6(const mib2_dccp6ConnEntry_t *,
    boolean_t, const mib2_transportMLPEntry_t *);

static void
dccp_report(const mib_item_t *item)
{
        mib2_dccpConnEntry_t            *dp;
        mib2_transportMLPEntry_t        **v4_attrs;
        mib2_transportMLPEntry_t        **v6_attrs;
        mib2_transportMLPEntry_t        **v4a;
        mib2_transportMLPEntry_t        **v6a;
        mib2_transportMLPEntry_t        *aptr;
        boolean_t                       print_hdr_once_v4 = B_TRUE;
        boolean_t                       print_hdr_once_v6 = B_TRUE;
        int                             jtemp = 0;

        if (!protocol_selected(IPPROTO_DCCP)) {
                return;
        }

        v4_attrs = family_selected(AF_INET) && RSECflag ?
            gather_attrs(item, MIB2_DCCP, MIB2_DCCP_CONN, dccpEntrySize) :
            NULL;
        v6_attrs = family_selected(AF_INET6) && RSECflag ?
            gather_attrs(item, MIB2_DCCP6, MIB2_DCCP6_CONN, dccp6EntrySize) :
            NULL;

        v4a = v4_attrs;
        v6a = v6_attrs;
        for (; item != NULL; item = item->next_item) {
                if (Xflag) {
                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
                        (void) printf("Group = %d, mib_id = %d, "
                            "length = %d, valp = 0x%p\n",
                            item->group, item->mib_id,
                            item->length, item->valp);
                }

                if (item->group == MIB2_DCCP) {
                        for (dp = (mib2_dccpConnEntry_t *)item->valp;
                            (char *)dp < (char *)item->valp + item->length;
                            dp = (mib2_dccpConnEntry_t *)((char *)dp +
```

```
5542                            dccpEntrySize)) {
5543                                aptr = v4a == NULL ? NULL : *v4a++;
5544                                print_hdr_once_v4 = dccp_report_item_v4(dp,
5545                                    print_hdr_once_v4, aptr);
5546                        }
5547                }
5548        }

5550        (void) fflush(stdout);

5552        if (v4_attrs != NULL) {
5553                free(v4_attrs);
5554        }
5555        if (v6_attrs != NULL) {
5556                free(v6_attrs);
5557        }
5558 }

5560 static boolean_t
5561 dccp_report_item_v4(const mib2_dccpConnEntry_t *dp, boolean_t first,
5562     const mib2_transportMLPEntry_t *attr)
5563 {
5564        char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5565        char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];

5567        if (first) {
5568                (void) printf(v4compat ? dccp_hdr_v4_compat : dccp_hdr_v4);
5569                (void) printf(Vflag ? dccp_hdr_v4_verbose : dccp_hdr_v4_normal);
5570        }

5572        (void) printf("%-20s %-20s %5u %6d %5u %6d %s\n",
5573            pr_ap(dp->dccpConnLocalAddress,
5574            dp->dccpConnLocalPort, "dccp", lname, sizeof (lname)),
5575            pr_ap(dp->dccpConnRemAddress,
5576            dp->dccpConnRemPort, "dccp", fname, sizeof (fname)),
5577            0,
5578            0,
5579            0,
5580            0,
5581            0);

5583        print_transport_label(attr);

5585        return (B_FALSE);
5586 }

5588 static boolean_t
5589 dccp_report_item_v6(const mib2_dccp6ConnEntry_t *dp, boolean_t first,
5590     const mib2_transportMLPEntry_t *attr)
5591 {
5592        return (B_FALSE);
5593 }

5595 #endif /* ! codereview */
5596 static char *
5597 plural(int n)
5598 {
5599        return (n != 1 ? "s" : "");
5600 }

5602 static char *
5603 pluraly(int n)
5604 {
5605        return (n != 1 ? "ies" : "y");
5606 }
```



```
5608 static char *
5609 plurales(int n)
5610 {
5611        return (n != 1 ? "es" : "");
5612 }

5614 static char *
5615 pktscale(n)
5616        int n;
5617 {
5618        static char buf[6];
5619        char t;

5621        if (n < 1024) {
5622                t = ' ';
5623        } else if (n < 1024 * 1024) {
5624                t = 'k';
5625                n /= 1024;
5626        } else if (n < 1024 * 1024 * 1024) {
5627                t = 'm';
5628                n /= 1024 * 1024;
5629        } else {
5630                t = 'g';
5631                n /= 1024 * 1024 * 1024;
5632        }

5634        (void) snprintf(buf, sizeof (buf), "%4u%c", n, t);
5635        return (buf);
5636 }

5638 /* --------------------- mrt_report (netstat -m) ------------------------- */

5640 static void
5641 mrt_report(mib_item_t *item)
5642 {
5643        int             jtemp = 0;
5644        struct vifctl   *vip;
5645        vifi_t          vifi;
5646        struct mfcctl   *mfccp;
5647        int             numvifs = 0;
5648        int             nmfc = 0;
5649        char            abuf[MAXHOSTNAMELEN + 1];

5651        if (!(family_selected(AF_INET)))
5652                return;

5654        /* 'for' loop 1: */
5655        for (; item; item = item->next_item) {
5656                if (Xflag) {
5657                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
5658                        (void) printf("Group = %d, mib_id = %d, "
5659                            "length = %d, valp = 0x%p\n",
5660                            item->group, item->mib_id, item->length,
5661                            item->valp);
5662                }
5663                if (item->group != EXPER_DVMRP)
5664                        continue; /* 'for' loop 1 */

5666                switch (item->mib_id) {

5668                case EXPER_DVMRP_VIF:
5669                        if (Xflag)
5670                                (void) printf("%u records for ipVifTable:\n",
5671                                    item->length/sizeof (struct vifctl));
5672                        if (item->length/sizeof (struct vifctl) == 0) {
5673                                (void) puts("\nVirtual Interface Table is "
```

```
5674                                    "empty");
5675                            break;
5676                    }

5678                    (void) puts("\nVirtual Interface Table\n"
5679                        " Vif Threshold Rate_Limit Local-Address"
5680                        "   Remote-Address     Pkt_in   Pkt_out");

5682                    /* 'for' loop 2: */
5683                    for (vip = (struct vifctl *)item->valp;
5684                        (char *)vip < (char *)item->valp + item->length;
5685                        /* LINTED: (note 1) */
5686                        vip = (struct vifctl *)((char *)vip +
5687                        vifctlSize)) {
5688                            if (vip->vifc_lcl_addr.s_addr == 0)
5689                                    continue; /* 'for' loop 2 */
5690                            /* numvifs = vip->vifc_vifi; */

5692                            numvifs++;
5693                            (void) printf("  %2u       %3u       "
5694                                "%4u %-15.15s",
5695                                vip->vifc_vifi,
5696                                vip->vifc_threshold,
5697                                vip->vifc_rate_limit,
5698                                pr_addr(vip->vifc_lcl_addr.s_addr,
5699                                abuf, sizeof (abuf)));
5700                            (void) printf(" %-15.15s  %8u  %8u\n",
5701                                (vip->vifc_flags & VIFF_TUNNEL) ?
5702                                pr_addr(vip->vifc_rmt_addr.s_addr,
5703                                abuf, sizeof (abuf)) : "",
5704                                vip->vifc_pkt_in,
5705                                vip->vifc_pkt_out);
5706                    } /* 'for' loop 2 ends */

5708                    (void) printf("Numvifs: %d\n", numvifs);
5709                    break;

5711            case EXPER_DVMRP_MRT:
5712                    if (Xflag)
5713                            (void) printf("%u records for ipMfcTable:\n",
5714                                item->length/sizeof (struct vifctl));
5715                    if (item->length/sizeof (struct vifctl) == 0) {
5716                            (void) puts("\nMulticast Forwarding Cache is "
5717                                "empty");
5718                            break;
5719                    }

5721                    (void) puts("\nMulticast Forwarding Cache\n"
5722                        "  Origin-Subnet                Mcastgroup      "
5723                        "# Pkts  In-Vif  Out-vifs/Forw-ttl");

5725                    for (mfccp = (struct mfcctl *)item->valp;
5726                        (char *)mfccp < (char *)item->valp + item->length;
5727                        /* LINTED: (note 1) */
5728                        mfccp = (struct mfcctl *)((char *)mfccp +
5729                        mfcctlSize)) {

5731                            nmfc++;
5732                            (void) printf("  %-30.15s",
5733                                pr_addr(mfccp->mfcc_origin.s_addr,
5734                                abuf, sizeof (abuf)));
5735                            (void) printf("%-15.15s  %6s  %3u    ",
5736                                pr_net(mfccp->mfcc_mcastgrp.s_addr,
5737                                mfccp->mfcc_mcastgrp.s_addr,
5738                                abuf, sizeof (abuf)),
5739                                pktscale((int)mfccp->mfcc_pkt_cnt),
```



```
5740                                    mfccp->mfcc_parent);

5742                            for (vifi = 0; vifi < MAXVIFS; ++vifi) {
5743                                    if (mfccp->mfcc_ttls[vifi]) {
5744                                            (void) printf("      %u (%u)",
5745                                                vifi,
5746                                                mfccp->mfcc_ttls[vifi]);
5747                                    }

5749                            }
5750                            (void) putchar('\n');
5751                    }
5752                    (void) printf("\nTotal no. of entries in cache: %d\n",
5753                        nmfc);
5754                    break;
5755            }
5756    } /* 'for' loop 1 ends */
5757    (void) putchar('\n');
5758    (void) fflush(stdout);
5759 }

5761 /*
5762  * Get the stats for the cache named 'name'.  If prefix != 0, then
5763  * interpret the name as a prefix, and sum up stats for all caches
5764  * named 'name*'.
5765  */
5766 static void
5767 kmem_cache_stats(char *title, char *name, int prefix, int64_t *total_bytes)
5768 {
5769         int len;
5770         int alloc;
5771         int64_t total_alloc = 0;
5772         int alloc_fail, total_alloc_fail = 0;
5773         int buf_size = 0;
5774         int buf_avail;
5775         int buf_total;
5776         int buf_max, total_buf_max = 0;
5777         int buf_inuse, total_buf_inuse = 0;
5778         kstat_t *ksp;
5779         char buf[256];

5781         len = prefix ? strlen(name) : 256;

5783         /* 'for' loop 1: */
5784         for (ksp = kc->kc_chain; ksp != NULL; ksp = ksp->ks_next) {

5786                 if (strcmp(ksp->ks_class, "kmem_cache") != 0)
5787                         continue; /* 'for' loop 1 */

5789                 /*
5790                  * Hack alert: because of the way streams messages are
5791                  * allocated, every constructed free dblk has an associated
5792                  * mblk.  From the allocator's viewpoint those mblks are
5793                  * allocated (because they haven't been freed), but from
5794                  * our viewpoint they're actually free (because they're
5795                  * not currently in use).  To account for this caching
5796                  * effect we subtract the total constructed free dblks
5797                  * from the total allocated mblks to derive mblks in use.
5798                  */
5799                 if (strcmp(name, "streams_mblk") == 0 &&
5800                     strncmp(ksp->ks_name, "streams_dblk", 12) == 0) {
5801                         (void) safe_kstat_read(kc, ksp, NULL);
5802                         total_buf_inuse -=
5803                             kstat_named_value(ksp, "buf_constructed");
5804                         continue; /* 'for' loop 1 */
5805                 }
```

```
5807                if (strncmp(ksp->ks_name, name, len) != 0)
5808                        continue; /* 'for' loop 1 */

5810                (void) safe_kstat_read(kc, ksp, NULL);

5812                alloc           = kstat_named_value(ksp, "alloc");
5813                alloc_fail      = kstat_named_value(ksp, "alloc_fail");
5814                buf_size        = kstat_named_value(ksp, "buf_size");
5815                buf_avail       = kstat_named_value(ksp, "buf_avail");
5816                buf_total       = kstat_named_value(ksp, "buf_total");
5817                buf_max         = kstat_named_value(ksp, "buf_max");
5818                buf_inuse       = buf_total - buf_avail;

5820                if (Vflag && prefix) {
5821                        (void) snprintf(buf, sizeof (buf), "%s%s", title,
5822                            ksp->ks_name + len);
5823                        (void) printf("    %-18s %6u %9u %11u %11u\n",
5824                            buf, buf_inuse, buf_max, alloc, alloc_fail);
5825                }

5827                total_alloc             += alloc;
5828                total_alloc_fail        += alloc_fail;
5829                total_buf_max           += buf_max;
5830                total_buf_inuse         += buf_inuse;
5831                *total_bytes            += (int64_t)buf_inuse * buf_size;
5832        } /* 'for' loop 1 ends */

5834        if (buf_size == 0) {
5835                (void) printf("%-22s [couldn't find statistics for %s]\n",
5836                    title, name);
5837                return;
5838        }

5840        if (Vflag && prefix)
5841                (void) snprintf(buf, sizeof (buf), "%s_total", title);
5842        else
5843                (void) snprintf(buf, sizeof (buf), "%s", title);

5845        (void) printf("%-22s %6d %9d %11lld %11d\n", buf,
5846            total_buf_inuse, total_buf_max, total_alloc, total_alloc_fail);
5847 }

5849 static void
5850 m_report(void)
5851 {
5852        int64_t total_bytes = 0;

5854        (void) puts("streams allocation:");
5855        (void) printf("%63s\n", "cumulative  allocation");
5856        (void) printf("%63s\n",
5857            "current   maximum       total    failures");

5859        kmem_cache_stats("streams",
5860            "stream_head_cache", 0, &total_bytes);
5861        kmem_cache_stats("queues", "queue_cache", 0, &total_bytes);
5862        kmem_cache_stats("mblk", "streams_mblk", 0, &total_bytes);
5863        kmem_cache_stats("dblk", "streams_dblk", 1, &total_bytes);
5864        kmem_cache_stats("linkblk", "linkinfo_cache", 0, &total_bytes);
5865        kmem_cache_stats("syncq", "syncq_cache", 0, &total_bytes);
5866        kmem_cache_stats("qband", "qband_cache", 0, &total_bytes);

5868        (void) printf("\n%lld Kbytes allocated for streams data\n",
5869            total_bytes / 1024);

5871        (void) putchar('\n');
```



```
5872        (void) fflush(stdout);
5873 }

5875 /* -------------------------------- */

5877 /*
5878  * Print an IPv4 address. Remove the matching part of the domain name
5879  * from the returned name.
5880  */
5881 static char *
5882 pr_addr(uint_t addr, char *dst, uint_t dstlen)
5883 {
5884        char                    *cp;
5885        struct hostent          *hp = NULL;
5886        static char             domain[MAXHOSTNAMELEN + 1];
5887        static boolean_t        first = B_TRUE;
5888        int                     error_num;

5890        if (first) {
5891                first = B_FALSE;
5892                if (sysinfo(SI_HOSTNAME, domain, MAXHOSTNAMELEN) != -1 &&
5893                    (cp = strchr(domain, '.'))) {
5894                        (void) strncpy(domain, cp + 1, sizeof (domain));
5895                } else
5896                        domain[0] = 0;
5897        }
5898        cp = NULL;
5899        if (!Nflag) {
5900                hp = getipnodebyaddr((char *)&addr, sizeof (uint_t), AF_INET,
5901                    &error_num);
5902                if (hp) {
5903                        if ((cp = strchr(hp->h_name, '.')) != NULL &&
5904                            strcasecmp(cp + 1, domain) == 0)
5905                                *cp = 0;
5906                        cp = hp->h_name;
5907                }
5908        }
5909        if (cp != NULL) {
5910                (void) strncpy(dst, cp, dstlen);
5911                dst[dstlen - 1] = 0;
5912        } else {
5913                (void) inet_ntop(AF_INET, (char *)&addr, dst, dstlen);
5914        }
5915        if (hp != NULL)
5916                freehostent(hp);
5917        return (dst);
5918 }

5920 /*
5921  * Print a non-zero IPv4 address.  Print "    --" if the address is zero.
5922  */
5923 static char *
5924 pr_addrnz(ipaddr_t addr, char *dst, uint_t dstlen)
5925 {
5926        if (addr == INADDR_ANY) {
5927                (void) strlcpy(dst, "    --", dstlen);
5928                return (dst);
5929        }
5930        return (pr_addr(addr, dst, dstlen));
5931 }

5933 /*
5934  * Print an IPv6 address. Remove the matching part of the domain name
5935  * from the returned name.
5936  */
5937 static char *
```

```
5938 pr_addr6(const struct in6_addr *addr, char *dst, uint_t dstlen)
5939 {
5940         char                    *cp;
5941         struct hostent          *hp = NULL;
5942         static char             domain[MAXHOSTNAMELEN + 1];
5943         static boolean_t        first = B_TRUE;
5944         int                     error_num;

5946         if (first) {
5947                 first = B_FALSE;
5948                 if (sysinfo(SI_HOSTNAME, domain, MAXHOSTNAMELEN) != -1 &&
5949                     (cp = strchr(domain, '.'))) {
5950                         (void) strncpy(domain, cp + 1, sizeof (domain));
5951                 } else
5952                         domain[0] = 0;
5953         }
5954         cp = NULL;
5955         if (!Nflag) {
5956                 hp = getipnodebyaddr((char *)addr,
5957                     sizeof (struct in6_addr), AF_INET6, &error_num);
5958                 if (hp) {
5959                         if ((cp = strchr(hp->h_name, '.')) != NULL &&
5960                             strcasecmp(cp + 1, domain) == 0)
5961                                 *cp = 0;
5962                         cp = hp->h_name;
5963                 }
5964         }
5965         if (cp != NULL) {
5966                 (void) strncpy(dst, cp, dstlen);
5967                 dst[dstlen - 1] = 0;
5968         } else {
5969                 (void) inet_ntop(AF_INET6, (void *)addr, dst, dstlen);
5970         }
5971         if (hp != NULL)
5972                 freehostent(hp);
5973         return (dst);
5974 }

5976 /* For IPv4 masks */
5977 static char *
5978 pr_mask(uint_t addr, char *dst, uint_t dstlen)
5979 {
5980         uint8_t *ip_addr = (uint8_t *)&addr;

5982         (void) snprintf(dst, dstlen, "%d.%d.%d.%d",
5983             ip_addr[0], ip_addr[1], ip_addr[2], ip_addr[3]);
5984         return (dst);
5985 }

5987 /*
5988  * For ipv6 masks format is : dest/mask
5989  * Does not print /128 to save space in printout. H flag carries this notion.
5990  */
5991 static char *
5992 pr_prefix6(const struct in6_addr *addr, uint_t prefixlen, char *dst,
5993     uint_t dstlen)
5994 {
5995         char *cp;

5997         if (IN6_IS_ADDR_UNSPECIFIED(addr) && prefixlen == 0) {
5998                 (void) strncpy(dst, "default", dstlen);
5999                 dst[dstlen - 1] = 0;
6000                 return (dst);
6001         }

6003         (void) pr_addr6(addr, dst, dstlen);
```



```
6004         if (prefixlen != IPV6_ABITS) {
6005                 /* How much room is left? */
6006                 cp = strchr(dst, '\0');
6007                 if (dst + dstlen > cp) {
6008                         dstlen -= (cp - dst);
6009                         (void) snprintf(cp, dstlen, "/%d", prefixlen);
6010                 }
6011         }
6012         return (dst);
6013 }

6015 /* Print IPv4 address and port */
6016 static char *
6017 pr_ap(uint_t addr, uint_t port, char *proto,
6018     char *dst, uint_t dstlen)
6019 {
6020         char *cp;

6022         if (addr == INADDR_ANY) {
6023                 (void) strncpy(dst, "      *", dstlen);
6024                 dst[dstlen - 1] = 0;
6025         } else {
6026                 (void) pr_addr(addr, dst, dstlen);
6027         }
6028         /* How much room is left? */
6029         cp = strchr(dst, '\0');
6030         if (dst + dstlen > cp + 1) {
6031                 *cp++ = '.';
6032                 dstlen -= (cp - dst);
6033                 dstlen--;
6034                 (void) portname(port, proto, cp, dstlen);
6035         }
6036         return (dst);
6037 }

6039 /* Print IPv6 address and port */
6040 static char *
6041 pr_ap6(const in6_addr_t *addr, uint_t port, char *proto,
6042     char *dst, uint_t dstlen)
6043 {
6044         char *cp;

6046         if (IN6_IS_ADDR_UNSPECIFIED(addr)) {
6047                 (void) strncpy(dst, "      *", dstlen);
6048                 dst[dstlen - 1] = 0;
6049         } else {
6050                 (void) pr_addr6(addr, dst, dstlen);
6051         }
6052         /* How much room is left? */
6053         cp = strchr(dst, '\0');
6054         if (dst + dstlen + 1 > cp) {
6055                 *cp++ = '.';
6056                 dstlen -= (cp - dst);
6057                 dstlen--;
6058                 (void) portname(port, proto, cp, dstlen);
6059         }
6060         return (dst);
6061 }

6063 /*
6064  * Return the name of the network whose address is given. The address is
6065  * assumed to be that of a net or subnet, not a host.
6066  */
6067 static char *
6068 pr_net(uint_t addr, uint_t mask, char *dst, uint_t dstlen)
6069 {
```

```
6070         char            *cp = NULL;
6071         struct netent   *np = NULL;
6072         struct hostent  *hp = NULL;
6073         uint_t          net;
6074         int             subnetshift;
6075         int             error_num;

6077         if (addr == INADDR_ANY && mask == INADDR_ANY) {
6078                 (void) strncpy(dst, "default", dstlen);
6079                 dst[dstlen - 1] = 0;
6080                 return (dst);
6081         }

6083         if (!Nflag && addr) {
6084                 if (mask == 0) {
6085                         if (IN_CLASSA(addr)) {
6086                                 mask = (uint_t)IN_CLASSA_NET;
6087                                 subnetshift = 8;
6088                         } else if (IN_CLASSB(addr)) {
6089                                 mask = (uint_t)IN_CLASSB_NET;
6090                                 subnetshift = 8;
6091                         } else {
6092                                 mask = (uint_t)IN_CLASSC_NET;
6093                                 subnetshift = 4;
6094                         }
6095                         /*
6096                          * If there are more bits than the standard mask
6097                          * would suggest, subnets must be in use. Guess at
6098                          * the subnet mask, assuming reasonable width subnet
6099                          * fields.
6100                          */
6101                         while (addr & ~mask)
6102                                 /* compiler doesn't sign extend! */
6103                                 mask = (mask | ((int)mask >> subnetshift));
6104                 }
6105                 net = addr & mask;
6106                 while ((mask & 1) == 0)
6107                         mask >>= 1, net >>= 1;
6108                 np = getnetbyaddr(net, AF_INET);
6109                 if (np && np->n_net == net)
6110                         cp = np->n_name;
6111                 else {
6112                         /*
6113                          * Look for subnets in hosts map.
6114                          */
6115                         hp = getipnodebyaddr((char *)&addr, sizeof (uint_t),
6116                             AF_INET, &error_num);
6117                         if (hp)
6118                                 cp = hp->h_name;
6119                 }
6120         }
6121         if (cp != NULL) {
6122                 (void) strncpy(dst, cp, dstlen);
6123                 dst[dstlen - 1] = 0;
6124         } else {
6125                 (void) inet_ntop(AF_INET, (char *)&addr, dst, dstlen);
6126         }
6127         if (hp != NULL)
6128                 freehostent(hp);
6129         return (dst);
6130 }

6132 /*
6133  * Return the name of the network whose address is given.
6134  * The address is assumed to be a host address.
6135  */
```



```
6136 static char *
6137 pr_netaddr(uint_t addr, uint_t mask, char *dst, uint_t dstlen)
6138 {
6139         char            *cp = NULL;
6140         struct netent   *np = NULL;
6141         struct hostent  *hp = NULL;
6142         uint_t          net;
6143         uint_t          netshifted;
6144         int             subnetshift;
6145         struct in_addr in;
6146         int             error_num;
6147         uint_t          nbo_addr = addr;        /* network byte order */

6149         addr = ntohl(addr);
6150         mask = ntohl(mask);
6151         if (addr == INADDR_ANY && mask == INADDR_ANY) {
6152                 (void) strncpy(dst, "default", dstlen);
6153                 dst[dstlen - 1] = 0;
6154                 return (dst);
6155         }

6157         /* Figure out network portion of address (with host portion = 0) */
6158         if (addr) {
6159                 /* Try figuring out mask if unknown (all 0s). */
6160                 if (mask == 0) {
6161                         if (IN_CLASSA(addr)) {
6162                                 mask = (uint_t)IN_CLASSA_NET;
6163                                 subnetshift = 8;
6164                         } else if (IN_CLASSB(addr)) {
6165                                 mask = (uint_t)IN_CLASSB_NET;
6166                                 subnetshift = 8;
6167                         } else {
6168                                 mask = (uint_t)IN_CLASSC_NET;
6169                                 subnetshift = 4;
6170                         }
6171                         /*
6172                          * If there are more bits than the standard mask
6173                          * would suggest, subnets must be in use. Guess at
6174                          * the subnet mask, assuming reasonable width subnet
6175                          * fields.
6176                          */
6177                         while (addr & ~mask)
6178                                 /* compiler doesn't sign extend! */
6179                                 mask = (mask | ((int)mask >> subnetshift));
6180                 }
6181                 net = netshifted = addr & mask;
6182                 while ((mask & 1) == 0)
6183                         mask >>= 1, netshifted >>= 1;
6184         }
6185         else
6186                 net = netshifted = 0;

6188         /* Try looking up name unless -n was specified. */
6189         if (!Nflag) {
6190                 np = getnetbyaddr(netshifted, AF_INET);
6191                 if (np && np->n_net == netshifted)
6192                         cp = np->n_name;
6193                 else {
6194                         /*
6195                          * Look for subnets in hosts map.
6196                          */
6197                         hp = getipnodebyaddr((char *)&nbo_addr, sizeof (uint_t),
6198                             AF_INET, &error_num);
6199                         if (hp)
6200                                 cp = hp->h_name;
6201                 }
```

```
6203                if (cp != NULL) {
6204                        (void) strncpy(dst, cp, dstlen);
6205                        dst[dstlen - 1] = 0;
6206                        if (hp != NULL)
6207                                freehostent(hp);
6208                        return (dst);
6209                }
6210                /*
6211                 * No name found for net: fallthru and return in decimal
6212                 * dot notation.
6213                 */
6214        }

6216        in.s_addr = htonl(net);
6217        (void) inet_ntop(AF_INET, (char *)&in, dst, dstlen);
6218        if (hp != NULL)
6219                freehostent(hp);
6220        return (dst);
6221 }

6223 /*
6224  * Return the filter mode as a string:
6225  *      1 => "INCLUDE"
6226  *      2 => "EXCLUDE"
6227  *      otherwise "<unknown>"
6228  */
6229 static char *
6230 fmodestr(uint_t fmode)
6231 {
6232        switch (fmode) {
6233        case 1:
6234                return ("INCLUDE");
6235        case 2:
6236                return ("EXCLUDE");
6237        default:
6238                return ("<unknown>");
6239        }
6240 }

6242 #define MAX_STRING_SIZE 256

6244 static const char *
6245 pr_secattr(const sec_attr_list_t *attrs)
6246 {
6247        int i;
6248        char buf[MAX_STRING_SIZE + 1], *cp;
6249        static char *sbuf;
6250        static size_t sbuf_len;
6251        struct rtsa_s rtsa;
6252        const sec_attr_list_t *aptr;

6254        if (!RSECflag || attrs == NULL)
6255                return ("");

6257        for (aptr = attrs, i = 1; aptr != NULL; aptr = aptr->sal_next)
6258                i += MAX_STRING_SIZE;
6259        if (i > sbuf_len) {
6260                cp = realloc(sbuf, i);
6261                if (cp == NULL) {
6262                        perror("realloc security attribute buffer");
6263                        return ("");
6264                }
6265                sbuf_len = i;
6266                sbuf = cp;
6267        }
```



```
6269        cp = sbuf;
6270        while (attrs != NULL) {
6271                const mib2_ipAttributeEntry_t *iae = attrs->sal_attr;

6273                /* note: effectively hard-coded in rtsa_keyword */
6274                rtsa.rtsa_mask = RTSA_CIPSO | RTSA_SLRANGE | RTSA_DOI;
6275                rtsa.rtsa_slrange = iae->iae_slrange;
6276                rtsa.rtsa_doi = iae->iae_doi;

6278                (void) snprintf(cp, MAX_STRING_SIZE,
6279                    "<%s>%s ", rtsa_to_str(&rtsa, buf, sizeof (buf)),
6280                    attrs->sal_next == NULL ? "" : ",");
6281                cp += strlen(cp);
6282                attrs = attrs->sal_next;
6283        }
6284        *cp = '\0';

6286        return (sbuf);
6287 }

6289 /*
6290  * Pretty print a port number. If the Nflag was
6291  * specified, use numbers instead of names.
6292  */
6293 static char *
6294 portname(uint_t port, char *proto, char *dst, uint_t dstlen)
6295 {
6296        struct servent *sp = NULL;

6298        if (!Nflag && port)
6299                sp = getservbyport(htons(port), proto);
6300        if (sp || port == 0)
6301                (void) snprintf(dst, dstlen, "%.*s", MAXHOSTNAMELEN,
6302                    sp ? sp->s_name : "*");
6303        else
6304                (void) snprintf(dst, dstlen, "%d", port);
6305        dst[dstlen - 1] = 0;
6306        return (dst);
6307 }

6309 /*PRINTFLIKE2*/
6310 void
6311 fail(int do_perror, char *message, ...)
6312 {
6313        va_list args;

6315        va_start(args, message);
6316        (void) fputs("netstat: ", stderr);
6317        (void) vfprintf(stderr, message, args);
6318        va_end(args);
6319        if (do_perror)
6320                (void) fprintf(stderr, ": %s", strerror(errno));
6321        (void) fputc('\n', stderr);
6322        exit(2);
6323 }

6325 /*
6326  * Return value of named statistic for given kstat_named kstat;
6327  * return 0LL if named statistic is not in list (use "ll" as a
6328  * type qualifier when printing 64-bit int's with printf() )
6329  */
6330 static uint64_t
6331 kstat_named_value(kstat_t *ksp, char *name)
6332 {
6333        kstat_named_t *knp;
```

```
6334         uint64_t value;

6336         if (ksp == NULL)
6337                 return (0LL);

6339         knp = kstat_data_lookup(ksp, name);
6340         if (knp == NULL)
6341                 return (0LL);

6343         switch (knp->data_type) {
6344         case KSTAT_DATA_INT32:
6345         case KSTAT_DATA_UINT32:
6346                 value = (uint64_t)(knp->value.ui32);
6347                 break;
6348         case KSTAT_DATA_INT64:
6349         case KSTAT_DATA_UINT64:
6350                 value = knp->value.ui64;
6351                 break;
6352         default:
6353                 value = 0LL;
6354                 break;
6355         }

6357         return (value);
6358 }

6360 kid_t
6361 safe_kstat_read(kstat_ctl_t *kc, kstat_t *ksp, void *data)
6362 {
6363         kid_t kstat_chain_id = kstat_read(kc, ksp, data);

6365         if (kstat_chain_id == -1)
6366                 fail(1, "kstat_read(%p, '%s') failed", (void *)kc,
6367                     ksp->ks_name);
6368         return (kstat_chain_id);
6369 }

6371 /*
6372  * Parse a list of IRE flag characters into a bit field.
6373  */
6374 static uint_t
6375 flag_bits(const char *arg)
6376 {
6377         const char *cp;
6378         uint_t val;

6380         if (*arg == '\0')
6381                 fatal(1, "missing flag list\n");

6383         val = 0;
6384         while (*arg != '\0') {
6385                 if ((cp = strchr(flag_list, *arg)) == NULL)
6386                         fatal(1, "%c: illegal flag\n", *arg);
6387                 val |= 1 << (cp - flag_list);
6388                 arg++;
6389         }
6390         return (val);
6391 }

6393 /*
6394  * Handle -f argument.  Validate input format, sort by keyword, and
6395  * save off digested results.
6396  */
6397 static void
6398 process_filter(char *arg)
6399 {
```



```
6400         int idx;
6401         int klen = 0;
6402         char *cp, *cp2;
6403         int val;
6404         filter_t *newf;
6405         struct hostent *hp;
6406         int error_num;
6407         uint8_t *ucp;
6408         int maxv;

6410         /* Look up the keyword first */
6411         if (strchr(arg, ':') == NULL) {
6412                 idx = FK_AF;
6413         } else {
6414                 for (idx = 0; idx < NFILTERKEYS; idx++) {
6415                         klen = strlen(filter_keys[idx]);
6416                         if (strncmp(filter_keys[idx], arg, klen) == 0 &&
6417                             arg[klen] == ':')
6418                                 break;
6419                 }
6420                 if (idx >= NFILTERKEYS)
6421                         fatal(1, "%s: unknown filter keyword\n", arg);

6423                 /* Advance past keyword and separator. */
6424                 arg += klen + 1;
6425         }

6427         if ((newf = malloc(sizeof (*newf))) == NULL) {
6428                 perror("filter");
6429                 exit(1);
6430         }
6431         switch (idx) {
6432         case FK_AF:
6433                 if (strcmp(arg, "inet") == 0) {
6434                         newf->u.f_family = AF_INET;
6435                 } else if (strcmp(arg, "inet6") == 0) {
6436                         newf->u.f_family = AF_INET6;
6437                 } else if (strcmp(arg, "unix") == 0) {
6438                         newf->u.f_family = AF_UNIX;
6439                 } else {
6440                         newf->u.f_family = strtol(arg, &cp, 0);
6441                         if (arg == cp || *cp != '\0')
6442                                 fatal(1, "%s: unknown address family.\n", arg);
6443                 }
6444                 break;

6446         case FK_OUTIF:
6447                 if (strcmp(arg, "none") == 0) {
6448                         newf->u.f_ifname = NULL;
6449                         break;
6450                 }
6451                 if (strcmp(arg, "any") == 0) {
6452                         newf->u.f_ifname = "";
6453                         break;
6454                 }
6455                 val = strtol(arg, &cp, 0);
6456                 if (val <= 0 || arg == cp || cp[0] != '\0') {
6457                         if ((val = if_nametoindex(arg)) == 0) {
6458                                 perror(arg);
6459                                 exit(1);
6460                         }
6461                 }
6462                 newf->u.f_ifname = arg;
6463                 break;

6465         case FK_DST:
```

```
6466                V4MASK_TO_V6(IP_HOST_MASK, newf->u.a.f_mask);
6467                if (strcmp(arg, "any") == 0) {
6468                        /* Special semantics; any address *but* zero */
6469                        newf->u.a.f_address = NULL;
6470                        (void) memset(&newf->u.a.f_mask, 0,
6471                            sizeof (newf->u.a.f_mask));
6472                        break;
6473                }
6474                if (strcmp(arg, "none") == 0) {
6475                        newf->u.a.f_address = NULL;
6476                        break;
6477                }
6478                if ((cp = strrchr(arg, '/')) != NULL)
6479                        *cp++ = '\0';
6480                hp = getipnodebyname(arg, AF_INET6, AI_V4MAPPED|AI_ALL,
6481                    &error_num);
6482                if (hp == NULL)
6483                        fatal(1, "%s: invalid or unknown host address\n", arg);
6484                newf->u.a.f_address = hp;
6485                if (cp == NULL) {
6486                        V4MASK_TO_V6(IP_HOST_MASK, newf->u.a.f_mask);
6487                } else {
6488                        val = strtol(cp, &cp2, 0);
6489                        if (cp != cp2 && cp2[0] == '\0') {
6490                                /*
6491                                 * If decode as "/n" works, then translate
6492                                 * into a mask.
6493                                 */
6494                                if (hp->h_addr_list[0] != NULL &&
6495                                    /* LINTED: (note 1) */
6496                                    IN6_IS_ADDR_V4MAPPED((in6_addr_t *)
6497                                    hp->h_addr_list[0])) {
6498                                        maxv = IP_ABITS;
6499                                } else {
6500                                        maxv = IPV6_ABITS;
6501                                }
6502                                if (val < 0 || val >= maxv)
6503                                        fatal(1, "%d: not in range 0 to %d\n",
6504                                            val, maxv - 1);
6505                                if (maxv == IP_ABITS)
6506                                        val += IPV6_ABITS - IP_ABITS;
6507                                ucp = newf->u.a.f_mask.s6_addr;
6508                                while (val >= 8)
6509                                        *ucp++ = 0xff, val -= 8;
6510                                *ucp++ = (0xff << (8 - val)) & 0xff;
6511                                while (ucp < newf->u.a.f_mask.s6_addr +
6512                                    sizeof (newf->u.a.f_mask.s6_addr))
6513                                        *ucp++ = 0;
6514                                /* Otherwise, try as numeric address */
6515                        } else if (inet_pton(AF_INET6,
6516                            cp, &newf->u.a.f_mask) <= 0) {
6517                                fatal(1, "%s: illegal mask format\n", cp);
6518                        }
6519                }
6520                break;

6522        case FK_FLAGS:
6523                if (*arg == '+') {
6524                        newf->u.f.f_flagset = flag_bits(arg + 1);
6525                        newf->u.f.f_flagclear = 0;
6526                } else if (*arg == '-') {
6527                        newf->u.f.f_flagset = 0;
6528                        newf->u.f.f_flagclear = flag_bits(arg + 1);
6529                } else {
6530                        newf->u.f.f_flagset = flag_bits(arg);
6531                        newf->u.f.f_flagclear = ~newf->u.f.f_flagset;
```



```
6532                }
6533                break;

6535        default:
6536                assert(0);
6537        }
6538        newf->f_next = filters[idx];
6539        filters[idx] = newf;
6540 }

6542 /* Determine if user wants this address family printed. */
6543 static boolean_t
6544 family_selected(int family)
6545 {
6546        const filter_t *fp;

6548        if (v4compat && family == AF_INET6)
6549                return (B_FALSE);
6550        if ((fp = filters[FK_AF]) == NULL)
6551                return (B_TRUE);
6552        while (fp != NULL) {
6553                if (fp->u.f_family == family)
6554                        return (B_TRUE);
6555                fp = fp->f_next;
6556        }
6557        return (B_FALSE);
6558 }

6560 /*
6561  * Convert the interface index to a string using the buffer 'ifname', which
6562  * must be at least LIFNAMSIZ bytes.  We first try to map it to name.  If that
6563  * fails (e.g., because we're inside a zone and it does not have access to
6564  * interface for the index in question), just return "if#<num>".
6565  */
6566 static char *
6567 ifindex2str(uint_t ifindex, char *ifname)
6568 {
6569        if (if_indextoname(ifindex, ifname) == NULL)
6570                (void) snprintf(ifname, LIFNAMSIZ, "if#%d", ifindex);

6572        return (ifname);
6573 }

6575 /*
6576  * print the usage line
6577  */
6578 static void
6579 usage(char *cmdname)
6580 {
6581        (void) fprintf(stderr, "usage: %s [-anv] [-f address_family] "
6582            "[-T d|u]\n", cmdname);
6583        (void) fprintf(stderr, "       %s [-n] [-f address_family] "
6584            "[-P protocol] [-T d|u] [-g | -p | -s [interval [count]]]\n",
6585            cmdname);
6586        (void) fprintf(stderr, "       %s -m [-v] [-T d|u] "
6587            "[interval [count]]\n", cmdname);
6588        (void) fprintf(stderr, "       %s -i [-I interface] [-an] "
6589            "[-f address_family] [-T d|u] [interval [count]]\n", cmdname);
6590        (void) fprintf(stderr, "       %s -r [-anv] "
6591            "[-f address_family|filter] [-T d|u]\n", cmdname);
6592        (void) fprintf(stderr, "       %s -M [-ns] [-f address_family] "
6593            "[-T d|u]\n", cmdname);
6594        (void) fprintf(stderr, "       %s -D [-I interface] "
6595            "[-f address_family] [-T d|u]\n", cmdname);
6596        exit(EXIT_FAILURE);
6597 }
```

```
6599 /*
6600  * fatal: print error message to stderr and
6601  * call exit(errcode)
6602  */
6603 /*PRINTFLIKE2*/
6604 static void
6605 fatal(int errcode, char *format, ...)
6606 {
6607         va_list argp;

6609         if (format == NULL)
6610                 return;

6612         va_start(argp, format);
6613         (void) vfprintf(stderr, format, argp);
6614         va_end(argp);

6616         exit(errcode);
6617 }
```

```
**********************************************************
   57122 Mon Jul  9 14:38:06 2012
new/usr/src/cmd/cmd-inet/usr.sbin/ipadm/ipadm.c
dccp: properties
**********************************************************
______unchanged_portion_omitted_

 634 /*
 635  * Properties to be displayed is in 'statep->sps_proplist'. If it is NULL,
 636  * for all the properties for the specified object, relevant information, will
 636  * for all the properties for the specified object, relavant information, will
 637  * be displayed. Otherwise, for the selected property set, display relevant
 638  * information
 639  */
 640 static void
 641 show_properties(void *arg, int prop_class)
 642 {
 643         show_prop_state_t       *statep = arg;
 644         nvlist_t                *nvl = statep->sps_proplist;
 645         uint_t                  proto = statep->sps_proto;
 646         nvpair_t                *curr_nvp;
 647         char                    *buf, *name;
 648         ipadm_status_t          status;

 650         /* allocate sufficient buffer to hold a property value */
 651         if ((buf = malloc(MAXPROPVALLEN)) == NULL)
 652                 die("insufficient memory");
 653         statep->sps_propval = buf;

 655         /* if no properties were specified, display all the properties */
 656         if (nvl == NULL) {
 657                 (void) ipadm_walk_proptbl(proto, prop_class, show_property,
 658                     statep);
 659         } else {
 660                 for (curr_nvp = nvlist_next_nvpair(nvl, NULL); curr_nvp;
 661                     curr_nvp = nvlist_next_nvpair(nvl, curr_nvp)) {
 662                         name = nvpair_name(curr_nvp);
 663                         status = ipadm_walk_prop(name, proto, prop_class,
 664                             show_property, statep);
 665                         if (status == IPADM_PROP_UNKNOWN)
 666                                 (void) show_property(statep, name, proto);
 667                 }
 668         }

 670         free(buf);
 671 }
______unchanged_portion_omitted_

 856 /*
 857  * Display information for all or specific protocol properties, either for a
 858  * given protocol or for supported protocols (IP/IPv4/IPv6/TCP/UDP/SCTP/DCCP)
 858  * given protocol or for supported protocols (IP/IPv4/IPv6/TCP/UDP/SCTP)
 859  */
 860 static void
 861 do_show_prop(int argc, char **argv, const char *use)
 862 {
 863         char                    option;
 864         nvlist_t                *proplist = NULL;
 865         char                    *fields_str = NULL;
 866         char                    *protostr;
 867         show_prop_state_t       state;
 868         ofmt_handle_t           ofmt;
 869         ofmt_status_t           oferr;
 870         uint_t                  ofmtflags = 0;
 871         uint_t                  proto;
 872         boolean_t               p_arg = _B_FALSE;
```



```
 874         opterr = 0;
 875         bzero(&state, sizeof (state));
 876         state.sps_propval = NULL;
 877         state.sps_parsable = _B_FALSE;
 878         state.sps_modprop = _B_TRUE;
 879         state.sps_status = state.sps_retstatus = IPADM_SUCCESS;
 880         while ((option = getopt_long(argc, argv, ":p:co:", show_prop_longopts,
 881             NULL)) != -1) {
 882                 switch (option) {
 883                 case 'p':
 884                         if (p_arg)
 885                                 die("-p must be specified once only");
 886                         p_arg = _B_TRUE;
 887                         if (ipadm_str2nvlist(optarg, &proplist,
 888                             IPADM_NORVAL) != 0)
 889                                 die("invalid protocol properties specified");
 890                         break;
 891                 case 'c':
 892                         state.sps_parsable = _B_TRUE;
 893                         break;
 894                 case 'o':
 895                         fields_str = optarg;
 896                         break;
 897                 default:
 898                         die_opterr(optopt, option, use);
 899                         break;
 900                 }
 901         }
 902         if (optind == argc - 1) {
 903                 protostr =  argv[optind];
 904                 if ((proto = ipadm_str2proto(protostr)) == MOD_PROTO_NONE)
 905                         die("invalid protocol '%s' specified", protostr);
 906                 state.sps_proto = proto;
 907         } else if (optind != argc) {
 908                 die("Usage: %s", use);
 909         } else {
 910                 if (p_arg)
 911                         die("protocol must be specified when "
 912                             "property name is used");
 913                 state.sps_proto = MOD_PROTO_NONE;
 914         }

 916         state.sps_proplist = proplist;

 918         if (state.sps_parsable)
 919                 ofmtflags |= OFMT_PARSABLE;
 920         else
 921                 ofmtflags |= OFMT_WRAP;
 922         oferr = ofmt_open(fields_str, modprop_fields, ofmtflags, 0, &ofmt);
 923         ipadm_ofmt_check(oferr, state.sps_parsable, ofmt);
 924         state.sps_ofmt = ofmt;

 926         /* handles all the errors */
 927         show_properties(&state, IPADMPROP_CLASS_MODULE);

 929         nvlist_free(proplist);
 930         ofmt_close(ofmt);

 932         if (state.sps_retstatus != IPADM_SUCCESS) {
 933                 ipadm_close(iph);
 934                 exit(EXIT_FAILURE);
 935         }
 936 }
______unchanged_portion_omitted_
```

```
**********************************************************
    2254 Mon Jul  9 14:38:06 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/Makefile
dccp: snoop, build system fixes
**********************************************************
```

```
   1 #
   2 # CDDL HEADER START
   3 #
   4 # The contents of this file are subject to the terms of the
   5 # Common Development and Distribution License (the "License").
   6 # You may not use this file except in compliance with the License.
   7 #
   8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9 # or http://www.opensolaris.org/os/licensing.
  10 # See the License for the specific language governing permissions
  11 # and limitations under the License.
  12 #
  13 # When distributing Covered Code, include this CDDL HEADER in each
  14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15 # If applicable, add the following below this CDDL HEADER, with the
  16 # fields enclosed by brackets "[]" replaced with your own identifying
  17 # information: Portions Copyright [yyyy] [name of copyright owner]
  18 #
  19 # CDDL HEADER END


  22 #
  23 # Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
  24 # Use is subject to license terms.
  25 #

  27 PROG=   snoop
  28 OBJS=   nfs4_xdr.o snoop.o snoop_aarp.o snoop_adsp.o snoop_aecho.o \
  29         snoop_apple.o snoop_arp.o snoop_atp.o snoop_bparam.o \
  30         snoop_bpdu.o \
  31         snoop_capture.o snoop_dccp.o snoop_dhcp.o snoop_dhcpv6.o \
  32         snoop_display.o snoop_dns.o snoop_ether.o \
  31         snoop_capture.o snoop_dhcp.o snoop_dhcpv6.o snoop_display.o \
  32         snoop_dns.o snoop_ether.o \
  33         snoop_filter.o snoop_http.o snoop_icmp.o snoop_igmp.o snoop_ip.o \
  34         snoop_ipaddr.o snoop_ipsec.o snoop_isis.o \
  35         snoop_ldap.o snoop_mip.o snoop_mount.o \
  36         snoop_nbp.o snoop_netbios.o snoop_nfs.o snoop_nfs3.o snoop_nfs4.o \
  37         snoop_nfs_acl.o snoop_nis.o snoop_nlm.o snoop_ntp.o \
  38         snoop_pf.o snoop_ospf.o snoop_ospf6.o snoop_pmap.o snoop_ppp.o \
  39         snoop_pppoe.o snoop_rip.o snoop_rip6.o snoop_rpc.o snoop_rpcprint.o \
  40         snoop_rpcsec.o snoop_rport.o snoop_rquota.o snoop_rstat.o snoop_rtmp.o \
  41         snoop_sctp.o snoop_slp.o snoop_smb.o snoop_socks.o snoop_solarnet.o \
  42         snoop_tcp.o snoop_tftp.o snoop_trill.o snoop_udp.o snoop_zip.o

  44 SRCS=   $(OBJS:.o=.c)
  45 HDRS=   snoop.h snoop_mip.h at.h snoop_ospf.h snoop_ospf6.h

  47 include ../../../Makefile.cmd

  49 CPPFLAGS += -I. -I$(SRC)/common/net/dhcp
  50 LDLIBS += -ldhcputil -ldlpi -lsocket -lnsl -ltsol
  51 LDFLAGS += $(MAPFILE.NGB:%=-M%)

  53 .KEEP_STATE:

  55 .PARALLEL: $(OBJS)

  57 all:    $(PROG)

  59 $(PROG): $(OBJS) $(MAPFILE.NGB)
```



```
  60         $(LINK.c) -o $@ $(OBJS) $(LDLIBS)
  61         $(POST_PROCESS)

  63 install:        all $(ROOTUSRSBINPROG)

  65 clean:
  66         $(RM) $(OBJS)

  68 lint:   lint_SRCS

  70 include ../../../Makefile.targ
```

****
12779 Mon Jul 9 14:38:06 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop.h
dccp: snoop, build system fixes
****
_____***unchanged_portion_omitted_***

```
121 /*
122  * Used to print nested protocol layers.  For example, an ip datagram included
123  * in an icmp error, or a PPP packet included in an LCP protocol reject..
124  */
125 extern char *prot_nest_prefix;

127 extern char *get_sum_line(void);
128 extern char *get_detail_line(int, int);
129 extern int want_packet(uchar_t *, int, int);
130 extern void set_vlan_id(int);
131 extern struct timeval prev_time;
132 extern void process_pkt(struct sb_hdr *, char *, int, int);
133 extern char *getflag(int, int, char *, char *);
134 extern void show_header(char *, char *, int);
135 extern void show_count(void);
136 extern void xdr_init(char *, int);
137 extern char *get_line(int, int);
138 extern int get_line_remain(void);
139 extern char getxdr_char(void);
140 extern char showxdr_char(char *);
141 extern uchar_t getxdr_u_char(void);
142 extern uchar_t showxdr_u_char(char *);
143 extern short getxdr_short(void);
144 extern short showxdr_short(char *);
145 extern ushort_t getxdr_u_short(void);
146 extern ushort_t showxdr_u_short(char *);
147 extern long getxdr_long(void);
148 extern long showxdr_long(char *);
149 extern ulong_t getxdr_u_long(void);
150 extern ulong_t showxdr_u_long(char *);
151 extern longlong_t getxdr_longlong(void);
152 extern longlong_t showxdr_longlong(char *);
153 extern u_longlong_t getxdr_u_longlong(void);
154 extern u_longlong_t showxdr_u_longlong(char *);
155 extern char *getxdr_opaque(char *, int);
156 extern char *getxdr_string(char *, int);
157 extern char *showxdr_string(int, char *);
158 extern char *getxdr_bytes(uint_t *);
159 extern void xdr_skip(int);
160 extern int getxdr_pos(void);
161 extern void setxdr_pos(int);
162 extern char *getxdr_context(char *, int);
163 extern char *showxdr_context(char *);
164 extern enum_t getxdr_enum(void);
165 extern void show_space(void);
166 extern void show_trailer(void);
167 extern char *getxdr_date(void);
168 extern char *showxdr_date(char *);
169 extern char *getxdr_date_ns(void);
170 char *format_time(int64_t sec, uint32_t nsec);
171 extern char *showxdr_date_ns(char *);
172 extern char *getxdr_hex(int);
173 extern char *showxdr_hex(int, char *);
174 extern bool_t getxdr_bool(void);
175 extern bool_t showxdr_bool(char *);
176 extern char *concat_args(char **, int);
177 extern int pf_compile(char *, int);
178 extern void compile(char *, int);
179 extern void load_names(char *);
```



```
180 extern void cap_write(struct sb_hdr *, char *, int, int);
181 extern void cap_open_read(const char *);
182 extern void cap_open_write(const char *);
183 extern void cap_read(int, int, int, void (*)(), int);
184 extern void cap_close(void);
185 extern boolean_t open_datalink(dlpi_handle_t *, const char *);
186 extern void init_datalink(dlpi_handle_t, ulong_t, ulong_t, struct timeval *,
187     struct Pf_ext_packetfilt *);
188 extern void net_read(dlpi_handle_t, size_t, int, void (*)(), int);
189 extern void click(int);
190 extern void show_pktinfo(int, int, char *, char *, struct timeval *,
191                 struct timeval *, int, int);
192 extern void show_line(char *);
193 /*PRINTFLIKE1*/
194 extern void show_printf(char *fmt, ...)
195     __PRINTFLIKE(1);
196 extern char *getxdr_time(void);
197 extern char *showxdr_time(char *);
198 extern char *addrtoname(int, const void *);
199 extern char *show_string(const char *, int, int);
200 extern void pr_err(const char *, ...);
201 extern void pr_errdlpi(dlpi_handle_t, const char *, int);
202 extern void check_retransmit(char *, ulong_t);
203 extern char *nameof_prog(int);
204 extern char *getproto(int);
205 extern uint8_t print_ipv6_extensions(int, uint8_t **, uint8_t *, int *, int *);
206 extern void protoprint(int, int, ulong_t, int, int, int, char *, int);
207 extern char *getportname(int, in_port_t);

209 extern void interpret_arp(int, struct arphdr *, int);
210 extern void interpret_bparam(int, int, int, int, int, char *, int);
211 extern void interpret_dns(int, int, const uchar_t *, int, int);
212 extern void interpret_mount(int, int, int, int, int, char *, int);
213 extern void interpret_nfs(int, int, int, int, int, char *, int);
214 extern void interpret_nfs3(int, int, int, int, int, char *, int);
215 extern void interpret_nfs4(int, int, int, int, int, char *, int);
216 extern void interpret_nfs4_cb(int, int, int, int, int, char *, int);
217 extern void interpret_nfs_acl(int, int, int, int, int, char *, int);
218 extern void interpret_nis(int, int, int, int, int, char *, int);
219 extern void interpret_nisbind(int, int, int, int, int, char *, int);
220 extern void interpret_nlm(int, int, int, int, int, char *, int);
221 extern void interpret_pmap(int, int, int, int, int, char *, int);
222 extern int interpret_reserved(int, int, in_port_t, in_port_t, char *, int);
223 extern void interpret_rquota(int, int, int, int, int, char *, int);
224 extern void interpret_rstat(int, int, int, int, int, char *, int);
225 extern void interpret_solarnet_fw(int, int, int, int, int, char *, int);
226 extern void interpret_ldap(int, char *, int, int, int);
227 extern void interpret_icmp(int, struct icmp *, int, int);
228 extern void interpret_icmpv6(int, icmp6_t *, int, int);
229 extern int interpret_ip(int, const struct ip *, int);
230 extern int interpret_ipv6(int, const ip6_t *, int);
231 extern int interpret_ppp(int, uchar_t *, int);
232 extern int interpret_pppoe(int, poep_t *, int);
233 struct tcphdr;
234 extern int interpret_tcp(int, struct tcphdr *, int, int);
235 struct udphdr;
236 extern int interpret_udp(int, struct udphdr *, int, int);
237 extern int interpret_esp(int, uint8_t *, int, int);
238 extern int interpret_ah(int, uint8_t *, int, int);
239 struct sctp_hdr;
240 extern void interpret_sctp(int, struct sctp_hdr *, int, int);
241 struct dccphdr;
242 extern int interpret_dccp(int, struct dccphdr *, int, int);
243 #endif /* ! codereview */
244 extern void interpret_mip_cntrlmsg(int, uchar_t *, int);
245 struct dhcp;
```

```
246 extern int interpret_dhcp(int, struct dhcp *, int);
247 extern int interpret_dhcpv6(int, const uint8_t *, int);
248 struct tftphdr;
249 extern int interpret_tftp(int, struct tftphdr *, int);
250 extern int interpret_http(int, char *, int);
251 struct ntpdata;
252 extern int interpret_ntp(int, struct ntpdata *, int);
253 extern void interpret_netbios_ns(int, uchar_t *, int);
254 extern void interpret_netbios_datagram(int, uchar_t *, int);
255 extern void interpret_netbios_ses(int, uchar_t *, int);
256 extern void interpret_slp(int, char *, int);
257 struct rip;
258 extern int interpret_rip(int, struct rip *, int);
259 struct rip6;
260 extern int interpret_rip6(int, struct rip6 *, int);
261 extern int interpret_socks_call(int, char *, int);
262 extern int interpret_socks_reply(int, char *, int);
263 extern int interpret_trill(int, struct ether_header **, char *, int *);
264 extern int interpret_isis(int, char *, int, boolean_t);
265 extern int interpret_bpdu(int, char *, int);
266 extern void init_ldap(void);
267 extern boolean_t arp_for_ether(char *, struct ether_addr *);
268 extern char *ether_ouiname(uint32_t);
269 extern char *tohex(char *p, int len);
270 extern char *printether(struct ether_addr *);
271 extern char *print_ethertype(int);
272 extern const char *arp_htype(int);
273 extern int valid_rpc(char *, int);

275 /*
276  * Describes characteristics of the Media Access Layer.
277  * The mac_type is one of the supported DLPI media
278  * types (see <sys/dlpi.h>).
279  * The mtu_size is the size of the largest frame.
280  * network_type_offset is where the network type
281  * is located in the link layer header.
282  * The header length is returned by a function to
283  * allow for variable header size - for ethernet it's
284  * just a constant 14 octets.
285  * The interpreter is the function that "knows" how
286  * to interpret the frame.
287  * try_kernel_filter tells snoop to first try a kernel
288  * filter (because the header size is fixed, or if it could
289  * be of variable size where the variable size is easy for a kernel
290  * filter to handle, for example, Ethernet and VLAN tags)
291  * and only use a user space filter if the filter expression
292  * cannot be expressed in kernel space.
293  */
294 typedef uint_t (interpreter_fn_t)(int, char *, int, int);
295 typedef uint_t (headerlen_fn_t)(char *, size_t);
296 typedef struct interface {
297         uint_t          mac_type;
298         uint_t          mtu_size;
299         uint_t          network_type_offset;
300         size_t          network_type_len;
301         uint_t          network_type_ip;
302         uint_t          network_type_ipv6;
303         headerlen_fn_t  *header_len;
304         interpreter_fn_t *interpreter;
305         boolean_t       try_kernel_filter;
306 } interface_t;

308 extern interface_t INTERFACES[], *interface;
309 extern char *dlc_header;
310 extern char *src_name, *dst_name;
311 extern char *prot_prefix;
```



```
312 extern char *prot_nest_prefix;
313 extern char *prot_title;

315 /* Keep track of how many nested IP headers we have. */
316 extern unsigned int encap_levels, total_encap_levels;

318 extern int quitting;
319 extern boolean_t Iflg, Pflg, rflg;

321 /*
322  * Global error recovery routine: used to reset snoop variables after
323  * catastrophic failure.
324  */
325 void snoop_recover(void);

327 /*
328  * Global alarm handler structure for managing multiple alarms within
329  * snoop.
330  */
331 typedef struct snoop_handler {
332         struct snoop_handler *s_next;           /* next alarm handler */
333         time_t s_time;                          /* time to fire */
334         void (*s_handler)();                    /* alarm handler */
335 } snoop_handler_t;

337 #define SNOOP_MAXRECOVER        20      /* maxium number of recoveries */
338 #define SNOOP_ALARM_GRAN        3       /* alarm() timeout multiplier */

340 /*
341  * Global alarm handler management routine.
342  */
343 extern int snoop_alarm(int s_sec, void (*s_handler)());

345 /*
346  * The next two definitions do not take into account the length
347  * of the underlying link header.  In order to use them, you must
348  * add link_header_len to them.  The reason it is not done here is
349  * that later these macros are used to initialize a table.
350  */
351 #define IPV4_TYPE_HEADER_OFFSET 9
352 #define IPV6_TYPE_HEADER_OFFSET 6

354 #ifdef __cplusplus
355 }
356 #endif

358 #endif  /* _SNOOP_H */
```

```
**********************************************************
    1672 Mon Jul  9 14:38:07 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_dccp.c
dccp: snoop, build system fixes
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>

#include <sys/socket.h>
#include <sys/sockio.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>
#include <netinet/dccp.h>
#include "snoop.h"

extern char *dlc_header;

int
interpret_dccp(int flags, struct dccphdr *dccp, int iplen, int fraglen)
{
        char *data;
        char *line;
        char *endline;

        if (flags & F_SUM) {
                line = get_sum_line();
                endline = line + MAXLINE;
                (void) snprintf(line, endline - line, "DCCP D=%d S=%d",
                    ntohs(dccp->dh_dport), ntohs(dccp->dh_sport));
        }

        return (fraglen);
```



```
}
#endif /* ! codereview */
```

```
**********************************************************
   63156 Mon Jul  9 14:38:07 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_filter.c
dccp: options and features
**********************************************************
______unchanged_portion_omitted_

1322 static match_type_t ether_match_types[] = {
1323         /*
1324          * Table initialized assuming Ethernet data link headers.
1325          * m_offset is an offset beyond the offset op, which is why
1326          * the offset is zero for when snoop needs to check an ethertype.
1327          */
1328         "ip",           0,  2, ETHERTYPE_IP,     -1,    OP_OFFSET_ETHERTYPE,
1329         "ip6",          0,  2, ETHERTYPE_IPV6,   -1,    OP_OFFSET_ETHERTYPE,
1330         "arp",          0,  2, ETHERTYPE_ARP,    -1,    OP_OFFSET_ETHERTYPE,
1331         "rarp",         0,  2, ETHERTYPE_REVARP, -1,    OP_OFFSET_ETHERTYPE,
1332         "pppoed",       0,  2, ETHERTYPE_PPPOED, -1,    OP_OFFSET_ETHERTYPE,
1333         "pppoes",       0,  2, ETHERTYPE_PPPOES, -1,    OP_OFFSET_ETHERTYPE,
1334         "tcp",          9,  1, IPPROTO_TCP,      0,     OP_OFFSET_LINK,
1335         "tcp",          6,  1, IPPROTO_TCP,      1,     OP_OFFSET_LINK,
1336         "udp",          9,  1, IPPROTO_UDP,      0,     OP_OFFSET_LINK,
1337         "udp",          6,  1, IPPROTO_UDP,      1,     OP_OFFSET_LINK,
1338         "icmp",         9,  1, IPPROTO_ICMP,     0,     OP_OFFSET_LINK,
1339         "icmp6",        6,  1, IPPROTO_ICMPV6,   1,     OP_OFFSET_LINK,
1340         "ospf",         9,  1, IPPROTO_OSPF,     0,     OP_OFFSET_LINK,
1341         "ospf",         6,  1, IPPROTO_OSPF,     1,     OP_OFFSET_LINK,
1342         "ip-in-ip",     9,  1, IPPROTO_ENCAP,    0,     OP_OFFSET_LINK,
1343         "esp",          9,  1, IPPROTO_ESP,      0,     OP_OFFSET_LINK,
1344         "esp",          6,  1, IPPROTO_ESP,      1,     OP_OFFSET_LINK,
1345         "ah",           9,  1, IPPROTO_AH,       0,     OP_OFFSET_LINK,
1346         "ah",           6,  1, IPPROTO_AH,       1,     OP_OFFSET_LINK,
1347         "sctp",         9,  1, IPPROTO_SCTP,     0,     OP_OFFSET_LINK,
1348         "sctp",         6,  1, IPPROTO_SCTP,     1,     OP_OFFSET_LINK,
1349         "dccp",         9,  1, IPPROTO_DCCP,     0,     OP_OFFSET_LINK,
1350         "dccp",         6,  1, IPPROTO_DCCP,     1,     OP_OFFSET_LINK,
1351 #endif /* ! codereview */
1352         0,              0,  0, 0,                0,     0
1353 };

1355 static match_type_t ipnet_match_types[] = {
1356         /*
1357          * Table initialized assuming Ethernet data link headers.
1358          * m_offset is an offset beyond the offset op, which is why
1359          * the offset is zero for when snoop needs to check an ethertype.
1360          */
1361         "ip",           0,  1, IPV4_VERSION,    -1,     OP_OFFSET_ETHERTYPE,
1362         "ip6",          0,  1, IPV6_VERSION,    -1,     OP_OFFSET_ETHERTYPE,
1363         "tcp",          9,  1, IPPROTO_TCP,      0,     OP_OFFSET_LINK,
1364         "tcp",          6,  1, IPPROTO_TCP,      1,     OP_OFFSET_LINK,
1365         "udp",          9,  1, IPPROTO_UDP,      0,     OP_OFFSET_LINK,
1366         "udp",          6,  1, IPPROTO_UDP,      1,     OP_OFFSET_LINK,
1367         "icmp",         9,  1, IPPROTO_ICMP,     0,     OP_OFFSET_LINK,
1368         "icmp6",        6,  1, IPPROTO_ICMPV6,   1,     OP_OFFSET_LINK,
1369         "ospf",         9,  1, IPPROTO_OSPF,     0,     OP_OFFSET_LINK,
1370         "ospf",         6,  1, IPPROTO_OSPF,     1,     OP_OFFSET_LINK,
1371         "ip-in-ip",     9,  1, IPPROTO_ENCAP,    0,     OP_OFFSET_LINK,
1372         "esp",          9,  1, IPPROTO_ESP,      0,     OP_OFFSET_LINK,
1373         "esp",          6,  1, IPPROTO_ESP,      1,     OP_OFFSET_LINK,
1374         "ah",           9,  1, IPPROTO_AH,       0,     OP_OFFSET_LINK,
1375         "ah",           6,  1, IPPROTO_AH,       1,     OP_OFFSET_LINK,
1376         "sctp",         9,  1, IPPROTO_SCTP,     0,     OP_OFFSET_LINK,
1377         "sctp",         6,  1, IPPROTO_SCTP,     1,     OP_OFFSET_LINK,
1378         "dccp",         9,  1, IPPROTO_DCCP,     0,     OP_OFFSET_LINK,
1379         "dccp",         6,  1, IPPROTO_DCCP,     1,     OP_OFFSET_LINK,
1380 #endif /* ! codereview */
```



```
1381         0,              0,  0, 0,                0,     0
1382 };

1384 static match_type_t iptun_match_types[] = {
1385         "ip",           0,  1, IPPROTO_ENCAP,   -1,     OP_OFFSET_ETHERTYPE,
1386         "ip6",          0,  1, IPPROTO_IPV6,    -1,     OP_OFFSET_ETHERTYPE,
1387         "tcp",          9,  1, IPPROTO_TCP,     0,      OP_OFFSET_LINK,
1388         "tcp",          6,  1, IPPROTO_TCP,     1,      OP_OFFSET_LINK,
1389         "udp",          9,  1, IPPROTO_UDP,     0,      OP_OFFSET_LINK,
1390         "udp",          6,  1, IPPROTO_UDP,     1,      OP_OFFSET_LINK,
1391         "icmp",         9,  1, IPPROTO_ICMP,    0,      OP_OFFSET_LINK,
1392         "icmp6",        6,  1, IPPROTO_ICMPV6,  1,      OP_OFFSET_LINK,
1393         "ospf",         9,  1, IPPROTO_OSPF,    0,      OP_OFFSET_LINK,
1394         "ospf",         6,  1, IPPROTO_OSPF,    1,      OP_OFFSET_LINK,
1395         "ip-in-ip",     9,  1, IPPROTO_ENCAP,   0,      OP_OFFSET_LINK,
1396         "esp",          9,  1, IPPROTO_ESP,     0,      OP_OFFSET_LINK,
1397         "esp",          6,  1, IPPROTO_ESP,     1,      OP_OFFSET_LINK,
1398         "ah",           9,  1, IPPROTO_AH,      0,      OP_OFFSET_LINK,
1399         "ah",           6,  1, IPPROTO_AH,      1,      OP_OFFSET_LINK,
1400         "sctp",         9,  1, IPPROTO_SCTP,    0,      OP_OFFSET_LINK,
1401         "sctp",         6,  1, IPPROTO_SCTP,    1,      OP_OFFSET_LINK,
1402         "dccp",         9,  1, IPPROTO_DCCP,    0,      OP_OFFSET_LINK,
1403         "dccp",         6,  1, IPPROTO_DCCP,    1,      OP_OFFSET_LINK,
1404 #endif /* ! codereview */
1405         0,              0,  0, 0,                0,     0
1406 };

1408 static void
1409 generate_check(match_type_t match_types[], int index, int type)
1410 {
1411         match_type_t *mtp = &match_types[index];
1412         /*
1413          * Note: this code assumes the above dependencies are
1414          * not cyclic.  This *should* always be true.
1415          */
1416         if (mtp->m_depend != -1)
1417                 generate_check(match_types, mtp->m_depend, type);

1419         emitop(mtp->m_optype);
1420         load_value(mtp->m_offset, mtp->m_size);
1421         load_const(mtp->m_value);
1422         emitop(OP_OFFSET_POP);

1424         emitop(OP_EQ);

1426         if (mtp->m_depend != -1)
1427                 emitop(OP_AND);
1428 }

1430 /*
1431  * Generate code based on the keyword argument.
1432  * This word is looked up in the match_types table
1433  * and checks a field within the packet for a given
1434  * value e.g. ether or ip type field.  The match
1435  * can also have a dependency on another entry e.g.
1436  * "tcp" requires that the packet also be "ip".
1437  */
1438 static int
1439 comparison(char *s)
1440 {
1441         unsigned int    i, n_checks = 0;
1442         match_type_t    *match_types;

1444         switch (interface->mac_type) {
1445         case DL_ETHER:
1446                 match_types = ether_match_types;
```

```
1447                 break;
1448         case DL_IPNET:
1449                 match_types = ipnet_match_types;
1450                 break;
1451         case DL_IPV4:
1452         case DL_IPV6:
1453         case DL_6TO4:
1454                 match_types = iptun_match_types;
1455                 break;
1456         default:
1457                 return (0);
1458         }

1460         for (i = 0; match_types[i].m_name != NULL; i++) {
1461                 if (strcmp(s, match_types[i].m_name) != 0)
1462                         continue;

1464                 n_checks++;
1465                 generate_check(match_types, i, interface->mac_type);
1466                 if (n_checks > 1)
1467                         emitop(OP_OR);
1468         }

1470         return (n_checks > 0);
1471 }

1473 enum direction { ANY, TO, FROM };
1474 enum direction dir;

1476 /*
1477  * Generate code to match an IP address.  The address
1478  * may be supplied either as a hostname or in dotted format.
1479  * For source packets both the IP source address and ARP
1480  * src are checked.
1481  * Note: we don't check packet type here - whether IP or ARP.
1482  * It's possible that we'll do an improper match.
1483  */
1484 static void
1485 ipaddr_match(enum direction which, char *hostname, int inet_type)
1486 {
1487         bool_t found_host;
1488         int m = 0, n = 0;
1489         uint_t *addr4ptr;
1490         uint_t addr4;
1491         struct in6_addr *addr6ptr;
1492         int h_addr_index;
1493         struct hostent *hp = NULL;
1494         int error_num = 0;
1495         boolean_t freehp = B_FALSE;
1496         boolean_t first = B_TRUE;

1498         /*
1499          * The addr4offset and addr6offset variables simplify the code which
1500          * generates the address comparison filter.  With these two variables,
1501          * duplicate code need not exist for the TO and FROM case.
1502          * A value of -1 describes the ANY case (TO and FROM).
1503          */
1504         int addr4offset;
1505         int addr6offset;

1507         found_host = 0;

1509         if (tokentype == ADDR_IP) {
1510                 hp = lgetipnodebyname(hostname, AF_INET, 0, &error_num);
1511                 if (hp == NULL) {
1512                         hp = getipnodebyname(hostname, AF_INET, 0, &error_num);
```



```
1513                         freehp = 1;
1514                 }
1515                 if (hp == NULL) {
1516                         if (error_num == TRY_AGAIN) {
1517                                 pr_err("couldn't resolve %s (try again later)",
1518                                     hostname);
1519                         } else {
1520                                 pr_err("couldn't resolve %s", hostname);
1521                         }
1522                 }
1523                 inet_type = IPV4_ONLY;
1524         } else if (tokentype == ADDR_IP6) {
1525                 hp = lgetipnodebyname(hostname, AF_INET6, 0, &error_num);
1526                 if (hp == NULL) {
1527                         hp = getipnodebyname(hostname, AF_INET6, 0, &error_num);
1528                         freehp = 1;
1529                 }
1530                 if (hp == NULL) {
1531                         if (error_num == TRY_AGAIN) {
1532                                 pr_err("couldn't resolve %s (try again later)",
1533                                     hostname);
1534                         } else {
1535                                 pr_err("couldn't resolve %s", hostname);
1536                         }
1537                 }
1538                 inet_type = IPV6_ONLY;
1539         } else {
1540                 /* Some hostname i.e. tokentype is ALPHA */
1541                 switch (inet_type) {
1542                 case IPV4_ONLY:
1543                         /* Only IPv4 address is needed */
1544                         hp = lgetipnodebyname(hostname, AF_INET, 0, &error_num);
1545                         if (hp == NULL) {
1546                                 hp = getipnodebyname(hostname, AF_INET, 0,
1547                                     &error_num);
1548                                 freehp = 1;
1549                         }
1550                         if (hp != NULL) {
1551                                 found_host = 1;
1552                         }
1553                         break;
1554                 case IPV6_ONLY:
1555                         /* Only IPv6 address is needed */
1556                         hp = lgetipnodebyname(hostname, AF_INET6, 0,
1557                             &error_num);
1558                         if (hp == NULL) {
1559                                 hp = getipnodebyname(hostname, AF_INET6, 0,
1560                                     &error_num);
1561                                 freehp = 1;
1562                         }
1563                         if (hp != NULL) {
1564                                 found_host = 1;
1565                         }
1566                         break;
1567                 case IPV4_AND_IPV6:
1568                         /* Both IPv4 and IPv6 are needed */
1569                         hp = lgetipnodebyname(hostname, AF_INET6,
1570                             AI_ALL | AI_V4MAPPED, &error_num);
1571                         if (hp == NULL) {
1572                                 hp = getipnodebyname(hostname, AF_INET6,
1573                                     AI_ALL | AI_V4MAPPED, &error_num);
1574                                 freehp = 1;
1575                         }
1576                         if (hp != NULL) {
1577                                 found_host = 1;
1578                         }
```

```
1579                        break;
1580                default:
1581                        found_host = 0;
1582                }

1584                if (!found_host) {
1585                        if (error_num == TRY_AGAIN) {
1586                                pr_err("could not resolve %s (try again later)",
1587                                    hostname);
1588                        } else {
1589                                pr_err("could not resolve %s", hostname);
1590                        }
1591                }
1592        }

1594        switch (which) {
1595        case TO:
1596                addr4offset = IPV4_DSTADDR_OFFSET;
1597                addr6offset = IPV6_DSTADDR_OFFSET;
1598                break;
1599        case FROM:
1600                addr4offset = IPV4_SRCADDR_OFFSET;
1601                addr6offset = IPV6_SRCADDR_OFFSET;
1602                break;
1603        case ANY:
1604                addr4offset = -1;
1605                addr6offset = -1;
1606                break;
1607        }

1609        /*
1610         * The code below generates the filter.
1611         */
1612        if (hp != NULL && hp->h_addrtype == AF_INET) {
1613                ethertype_match(interface->network_type_ip);
1614                emitop(OP_BRFL);
1615                n = chain(n);
1616                emitop(OP_OFFSET_LINK);
1617                h_addr_index = 0;
1618                addr4ptr = (uint_t *)hp->h_addr_list[h_addr_index];
1619                while (addr4ptr != NULL) {
1620                        if (addr4offset == -1) {
1621                                compare_addr_v4(IPV4_SRCADDR_OFFSET, 4,
1622                                    *addr4ptr);
1623                                emitop(OP_BRTR);
1624                                m = chain(m);
1625                                compare_addr_v4(IPV4_DSTADDR_OFFSET, 4,
1626                                    *addr4ptr);
1627                        } else {
1628                                compare_addr_v4(addr4offset, 4, *addr4ptr);
1629                        }
1630                        addr4ptr = (uint_t *)hp->h_addr_list[++h_addr_index];
1631                        if (addr4ptr != NULL) {
1632                                emitop(OP_BRTR);
1633                                m = chain(m);
1634                        }
1635                }
1636                if (m != 0) {
1637                        resolve_chain(m);
1638                }
1639                emitop(OP_OFFSET_POP);
1640                resolve_chain(n);
1641        } else {
1642                /* first pass: IPv4 addresses */
1643                h_addr_index = 0;
1644                addr6ptr = (struct in6_addr *)hp->h_addr_list[h_addr_index];
```



```
1645                first = B_TRUE;
1646                while (addr6ptr != NULL) {
1647                        if (IN6_IS_ADDR_V4MAPPED(addr6ptr)) {
1648                                if (first) {
1649                                        ethertype_match(
1650                                            interface->network_type_ip);
1651                                        emitop(OP_BRFL);
1652                                        n = chain(n);
1653                                        emitop(OP_OFFSET_LINK);
1654                                        first = B_FALSE;
1655                                } else {
1656                                        emitop(OP_BRTR);
1657                                        m = chain(m);
1658                                }
1659                                IN6_V4MAPPED_TO_INADDR(addr6ptr,
1660                                    (struct in_addr *)&addr4);
1661                                if (addr4offset == -1) {
1662                                        compare_addr_v4(IPV4_SRCADDR_OFFSET, 4,
1663                                            addr4);
1664                                        emitop(OP_BRTR);
1665                                        m = chain(m);
1666                                        compare_addr_v4(IPV4_DSTADDR_OFFSET, 4,
1667                                            addr4);
1668                                } else {
1669                                        compare_addr_v4(addr4offset, 4, addr4);
1670                                }
1671                        }
1672                        addr6ptr = (struct in6_addr *)
1673                            hp->h_addr_list[++h_addr_index];
1674                }
1675                /* second pass: IPv6 addresses */
1676                h_addr_index = 0;
1677                addr6ptr = (struct in6_addr *)hp->h_addr_list[h_addr_index];
1678                first = B_TRUE;
1679                while (addr6ptr != NULL) {
1680                        if (!IN6_IS_ADDR_V4MAPPED(addr6ptr)) {
1681                                if (first) {
1682                                        /*
1683                                         * bypass check for IPv6 addresses
1684                                         * when we have an IPv4 packet
1685                                         */
1686                                        if (n != 0) {
1687                                                emitop(OP_BRTR);
1688                                                m = chain(m);
1689                                                emitop(OP_BRFL);
1690                                                m = chain(m);
1691                                                resolve_chain(n);
1692                                                n = 0;
1693                                        }
1694                                        ethertype_match(
1695                                            interface->network_type_ipv6);
1696                                        emitop(OP_BRFL);
1697                                        n = chain(n);
1698                                        emitop(OP_OFFSET_LINK);
1699                                        first = B_FALSE;
1700                                } else {
1701                                        emitop(OP_BRTR);
1702                                        m = chain(m);
1703                                }
1704                                if (addr6offset == -1) {
1705                                        compare_addr_v6(IPV6_SRCADDR_OFFSET,
1706                                            16, *addr6ptr);
1707                                        emitop(OP_BRTR);
1708                                        m = chain(m);
1709                                        compare_addr_v6(IPV6_DSTADDR_OFFSET,
1710                                            16, *addr6ptr);
```

```
1711                                } else {
1712                                        compare_addr_v6(addr6offset, 16,
1713                                            *addr6ptr);
1714                                }
1715                        }
1716                        addr6ptr = (struct in6_addr *)
1717                            hp->h_addr_list[++h_addr_index];
1718                }
1719                if (m != 0) {
1720                        resolve_chain(m);
1721                }
1722                emitop(OP_OFFSET_POP);
1723                resolve_chain(n);
1724        }

1726        /* only free struct hostent returned by getipnodebyname() */
1727        if (freehp) {
1728                freehostent(hp);
1729        }
1730 }

1732 /*
1733  * Match on zoneid. The arg zone passed in is in network byte order.
1734  */
1735 static void
1736 zone_match(enum direction which, uint32_t zone)
1737 {

1739        switch (which) {
1740        case TO:
1741                compare_value_zone(IPNET_DSTZONE_OFFSET, zone);
1742                break;
1743        case FROM:
1744                compare_value_zone(IPNET_SRCZONE_OFFSET, zone);
1745                break;
1746        case ANY:
1747                compare_value_zone(IPNET_SRCZONE_OFFSET, zone);
1748                compare_value_zone(IPNET_DSTZONE_OFFSET, zone);
1749                emitop(OP_OR);
1750        }
1751 }

1753 /*
1754  * Generate code to match an AppleTalk address.  The address
1755  * must be given as two numbers with a dot between
1756  *
1757  */
1758 static void
1759 ataddr_match(enum direction which, char *hostname)
1760 {
1761        uint_t net;
1762        uint_t node;
1763        uint_t m, n;

1765        sscanf(hostname, "%u.%u", &net, &node);

1767        emitop(OP_OFFSET_LINK);
1768        switch (which) {
1769        case TO:
1770                compare_value(AT_DST_NET_OFFSET, 2, net);
1771                emitop(OP_BRFL);
1772                m = chain(0);
1773                compare_value(AT_DST_NODE_OFFSET, 1, node);
1774                resolve_chain(m);
1775                break;
1776        case FROM:
```



```
1777                compare_value(AT_SRC_NET_OFFSET, 2, net);
1778                emitop(OP_BRFL);
1779                m = chain(0);
1780                compare_value(AT_SRC_NODE_OFFSET, 1, node);
1781                resolve_chain(m);
1782                break;
1783        case ANY:
1784                compare_value(AT_DST_NET_OFFSET, 2, net);
1785                emitop(OP_BRFL);
1786                m = chain(0);
1787                compare_value(AT_DST_NODE_OFFSET, 1, node);
1788                resolve_chain(m);
1789                emitop(OP_BRTR);
1790                n = chain(0);
1791                compare_value(AT_SRC_NET_OFFSET, 2, net);
1792                emitop(OP_BRFL);
1793                m = chain(0);
1794                compare_value(AT_SRC_NODE_OFFSET, 1, node);
1795                resolve_chain(m);
1796                resolve_chain(n);
1797                break;
1798        }
1799        emitop(OP_OFFSET_POP);
1800 }

1802 /*
1803  * Compare ethernet addresses. The address may
1804  * be provided either as a hostname or as a
1805  * 6 octet colon-separated address.
1806  */
1807 static void
1808 etheraddr_match(enum direction which, char *hostname)
1809 {
1810        uint_t addr;
1811        ushort_t *addrp;
1812        int to_offset, from_offset;
1813        struct ether_addr e, *ep = NULL;
1814        int m;

1816        /*
1817         * First, check the interface type for whether src/dest address
1818         * is determinable; if not, retreat early.
1819         */
1820        switch (interface->mac_type) {
1821        case DL_ETHER:
1822                from_offset = ETHERADDRL;
1823                to_offset = 0;
1824                break;

1826        case DL_IB:
1827                /*
1828                 * If an ethernet address is attempted to be used
1829                 * on an IPoIB interface, flag error. Link address
1830                 * based filtering is unsupported on IPoIB, so there
1831                 * is no ipibaddr_match() or parsing support for IPoIB
1832                 * 20 byte link addresses.
1833                 */
1834                pr_err("filter option unsupported on media");
1835                break;

1837        case DL_FDDI:
1838                from_offset = 7;
1839                to_offset = 1;
1840                break;

1842        default:
```

```
1843                 /*
1844                  * Where do we find "ether" address for FDDI & TR?
1845                  * XXX can improve?  ~sparker
1846                  */
1847                 load_const(1);
1848                 return;
1849         }

1851         if (isxdigit(*hostname))
1852                 ep = ether_aton(hostname);
1853         if (ep == NULL) {
1854                 if (ether_hostton(hostname, &e))
1855                         if (!arp_for_ether(hostname, &e))
1856                                 pr_err("cannot obtain ether addr for %s",
1857                                     hostname);
1858                 ep = &e;
1859         }
1860         memcpy(&addr, (ushort_t *)ep, 4);
1861         addrp = (ushort_t *)ep + 2;

1863         emitop(OP_OFFSET_ZERO);
1864         switch (which) {
1865         case TO:
1866                 compare_value(to_offset, 4, ntohl(addr));
1867                 emitop(OP_BRFL);
1868                 m = chain(0);
1869                 compare_value(to_offset + 4, 2, ntohs(*addrp));
1870                 resolve_chain(m);
1871                 break;
1872         case FROM:
1873                 compare_value(from_offset, 4, ntohl(addr));
1874                 emitop(OP_BRFL);
1875                 m = chain(0);
1876                 compare_value(from_offset + 4, 2, ntohs(*addrp));
1877                 resolve_chain(m);
1878                 break;
1879         case ANY:
1880                 compare_value(to_offset, 4, ntohl(addr));
1881                 compare_value(to_offset + 4, 2, ntohs(*addrp));
1882                 emitop(OP_AND);
1883                 emitop(OP_BRTR);
1884                 m = chain(0);

1886                 compare_value(from_offset, 4, ntohl(addr));
1887                 compare_value(from_offset + 4, 2, ntohs(*addrp));
1888                 emitop(OP_AND);
1889                 resolve_chain(m);
1890                 break;
1891         }
1892         emitop(OP_OFFSET_POP);
1893 }

1895 static void
1896 ethertype_match(int val)
1897 {
1898         int ether_offset = interface->network_type_offset;

1900         /*
1901          * If the user is interested in ethertype VLAN,
1902          * then we need to set the offset to the beginning of the packet.
1903          * But if the user is interested in another ethertype,
1904          * such as IPv4, then we need to take into consideration
1905          * the fact that the packet might be VLAN tagged.
1906          */
1907         if (interface->mac_type == DL_ETHER ||
1908             interface->mac_type == DL_CSMACD) {
```



```
1909                 if (val != ETHERTYPE_VLAN) {
1910                         /*
1911                          * OP_OFFSET_ETHERTYPE puts us at the ethertype
1912                          * field whether or not there is a VLAN tag,
1913                          * so ether_offset goes to zero if we get here.
1914                          */
1915                         emitop(OP_OFFSET_ETHERTYPE);
1916                         ether_offset = 0;
1917                 } else {
1918                         emitop(OP_OFFSET_ZERO);
1919                 }
1920         }
1921         compare_value(ether_offset, interface->network_type_len, val);
1922         if (interface->mac_type == DL_ETHER ||
1923             interface->mac_type == DL_CSMACD) {
1924                 emitop(OP_OFFSET_POP);
1925         }
1926 }

1928 /*
1929  * Match a network address.  The host part
1930  * is masked out.  The network address may
1931  * be supplied either as a netname or in
1932  * IP dotted format.  The mask to be used
1933  * for the comparison is assumed from the
1934  * address format (see comment below).
1935  */
1936 static void
1937 netaddr_match(enum direction which, char *netname)
1938 {
1939         uint_t addr;
1940         uint_t mask = 0xff000000;
1941         uint_t m;
1942         struct netent *np;

1944         if (isdigit(*netname)) {
1945                 addr = inet_network(netname);
1946         } else {
1947                 np = getnetbyname(netname);
1948                 if (np == NULL)
1949                         pr_err("net %s not known", netname);
1950                 addr = np->n_net;
1951         }

1953         /*
1954          * Left justify the address and figure
1955          * out a mask based on the supplied address.
1956          * Set the mask according to the number of zero
1957          * low-order bytes.
1958          * Note: this works only for whole octet masks.
1959          */
1960         if (addr) {
1961                 while ((addr & ~mask) != 0) {
1962                         mask |= (mask >> 8);
1963                 }
1964         }

1966         emitop(OP_OFFSET_LINK);
1967         switch (which) {
1968         case TO:
1969                 compare_value_mask(16, 4, addr, mask);
1970                 break;
1971         case FROM:
1972                 compare_value_mask(12, 4, addr, mask);
1973                 break;
1974         case ANY:
```

```
1975                compare_value_mask(12, 4, addr, mask);
1976                emitop(OP_BRTR);
1977                m = chain(0);
1978                compare_value_mask(16, 4, addr, mask);
1979                resolve_chain(m);
1980                break;
1981        }
1982        emitop(OP_OFFSET_POP);
1983 }

1985 /*
1986  * Match either a UDP or TCP port number.
1987  * The port number may be provided either as
1988  * port name as listed in /etc/services ("nntp") or as
1989  * the port number itself (2049).
1990  */
1991 static void
1992 port_match(enum direction which, char *portname)
1993 {
1994        struct servent *sp;
1995        uint_t m, port;

1997        if (isdigit(*portname)) {
1998                port = atoi(portname);
1999        } else {
2000                sp = getservbyname(portname, NULL);
2001                if (sp == NULL)
2002                        pr_err("invalid port number or name: %s", portname);
2003                port = ntohs(sp->s_port);
2004        }

2006        emitop(OP_OFFSET_IP);

2008        switch (which) {
2009        case TO:
2010                compare_value(2, 2, port);
2011                break;
2012        case FROM:
2013                compare_value(0, 2, port);
2014                break;
2015        case ANY:
2016                compare_value(2, 2, port);
2017                emitop(OP_BRTR);
2018                m = chain(0);
2019                compare_value(0, 2, port);
2020                resolve_chain(m);
2021                break;
2022        }
2023        emitop(OP_OFFSET_POP);
2024 }

2026 /*
2027  * Generate code to match packets with a specific
2028  * RPC program number.  If the progname is a name
2029  * it is converted to a number via /etc/rpc.
2030  * The program version and/or procedure may be provided
2031  * as extra qualifiers.
2032  */
2033 static void
2034 rpc_match_prog(enum direction which, char *progname, int vers, int proc)
2035 {
2036        struct rpcent *rpc;
2037        uint_t prog;
2038        uint_t m, n;

2040        if (isdigit(*progname)) {
```



```
2041                prog = atoi(progname);
2042        } else {
2043                rpc = (struct rpcent *)getrpcbyname(progname);
2044                if (rpc == NULL)
2045                        pr_err("invalid program name: %s", progname);
2046                prog = rpc->r_number;
2047        }

2049        emitop(OP_OFFSET_RPC);
2050        emitop(OP_BRFL);
2051        n = chain(0);

2053        compare_value(12, 4, prog);
2054        emitop(OP_BRFL);
2055        m = chain(0);
2056        if (vers >= 0) {
2057                compare_value(16, 4, vers);
2058                emitop(OP_BRFL);
2059                m = chain(m);
2060        }
2061        if (proc >= 0) {
2062                compare_value(20, 4, proc);
2063                emitop(OP_BRFL);
2064                m = chain(m);
2065        }

2067        switch (which) {
2068        case TO:
2069                compare_value(4, 4, CALL);
2070                emitop(OP_BRFL);
2071                m = chain(m);
2072                break;
2073        case FROM:
2074                compare_value(4, 4, REPLY);
2075                emitop(OP_BRFL);
2076                m = chain(m);
2077                break;
2078        }
2079        resolve_chain(m);
2080        resolve_chain(n);
2081        emitop(OP_OFFSET_POP);
2082 }

2084 /*
2085  * Generate code to parse a field specification
2086  * and load the value of the field from the packet
2087  * onto the operand stack.
2088  * The field offset may be specified relative to the
2089  * beginning of the ether header, IP header, UDP header,
2090  * or TCP header.  An optional size specification may
2091  * be provided following a colon.  If no size is given
2092  * one byte is assumed e.g.
2093  *
2094  *      ether[0]        The first byte of the ether header
2095  *      ip[2:2]         The second 16 bit field of the IP header
2096  */
2097 static void
2098 load_field()
2099 {
2100        int size = 1;
2101        int s;


2104        if (EQ("ether"))
2105                emitop(OP_OFFSET_ZERO);
2106        else if (EQ("ip") || EQ("ip6") || EQ("pppoed") || EQ("pppoes"))
```

```
2107                    emitop(OP_OFFSET_LINK);
2108            else if (EQ("udp") || EQ("tcp") || EQ("icmp") || EQ("ip-in-ip") ||
2109                EQ("ah") || EQ("esp"))
2110                    emitop(OP_OFFSET_IP);
2111            else
2112                    pr_err("invalid field type");
2113            next();
2114            s = opstack;
2115            expression();
2116            if (opstack != s + 1)
2117                    pr_err("invalid field offset");
2118            opstack--;
2119            if (*token == ':') {
2120                    next();
2121                    if (tokentype != NUMBER)
2122                            pr_err("field size expected");
2123                    size = tokenval;
2124                    if (size != 1 && size != 2 && size != 4)
2125                            pr_err("field size invalid");
2126                    next();
2127            }
2128            if (*token != ']')
2129                    pr_err("right bracket expected");

2131            load_value(-1, size);
2132            emitop(OP_OFFSET_POP);
2133    }

2135    /*
2136     * Check that the operand stack
2137     * contains n arguments
2138     */
2139    static void
2140    checkstack(int numargs)
2141    {
2142            if (opstack != numargs)
2143                    pr_err("invalid expression at \"%s\".", token);
2144    }

2146    static void
2147    primary()
2148    {
2149            int m, m2, s;

2151            for (;;) {
2152                    if (tokentype == FIELD) {
2153                            load_field();
2154                            opstack++;
2155                            next();
2156                            break;
2157                    }

2159                    if (comparison(token)) {
2160                            opstack++;
2161                            next();
2162                            break;
2163                    }

2165                    if (EQ("not") || EQ("!")) {
2166                            next();
2167                            s = opstack;
2168                            primary();
2169                            checkstack(s + 1);
2170                            emitop(OP_NOT);
2171                            break;
2172                    }
```



```
2174                    if (EQ("(")) {
2175                            next();
2176                            s = opstack;
2177                            expression();
2178                            checkstack(s + 1);
2179                            if (!EQ(")"))
2180                                    pr_err("right paren expected");
2181                            next();
2182                    }

2184                    if (EQ("to") || EQ("dst")) {
2185                            dir = TO;
2186                            next();
2187                            continue;
2188                    }

2190                    if (EQ("from") || EQ("src")) {
2191                            dir = FROM;
2192                            next();
2193                            continue;
2194                    }

2196                    if (EQ("ether")) {
2197                            eaddr = 1;
2198                            next();
2199                            continue;
2200                    }

2202                    if (EQ("proto")) {
2203                            next();
2204                            if (tokentype != NUMBER)
2205                                    pr_err("IP proto type expected");
2206                            emitop(OP_OFFSET_LINK);
2207                            compare_value(IPV4_TYPE_HEADER_OFFSET, 1, tokenval);
2208                            emitop(OP_OFFSET_POP);
2209                            opstack++;
2210                            next();
2211                            continue;
2212                    }

2214                    if (EQ("broadcast")) {
2215                            /*
2216                             * Be tricky: FDDI ether dst address begins at
2217                             * byte one.  Since the address is really six
2218                             * bytes long, this works for FDDI & ethernet.
2219                             * XXX - Token ring?
2220                             */
2221                            emitop(OP_OFFSET_ZERO);
2222                            if (interface->mac_type == DL_IB)
2223                                    pr_err("filter option unsupported on media");
2224                            compare_value(1, 4, 0xffffffff);
2225                            emitop(OP_OFFSET_POP);
2226                            opstack++;
2227                            next();
2228                            break;
2229                    }

2231                    if (EQ("multicast")) {
2232                            /* XXX Token ring? */
2233                            emitop(OP_OFFSET_ZERO);
2234                            if (interface->mac_type == DL_FDDI) {
2235                                    compare_value_mask(1, 1, 0x01, 0x01);
2236                            } else if (interface->mac_type == DL_IB) {
2237                                    pr_err("filter option unsupported on media");
2238                            } else {
```

```
2239                            compare_value_mask(0, 1, 0x01, 0x01);
2240                    }
2241                    emitop(OP_OFFSET_POP);
2242                    opstack++;
2243                    next();
2244                    break;
2245            }

2247            if (EQ("decnet")) {
2248                    /* XXX Token ring? */
2249                    if (interface->mac_type == DL_FDDI) {
2250                            load_value(19, 2);      /* ether type */
2251                            load_const(0x6000);
2252                            emitop(OP_GE);
2253                            emitop(OP_BRFL);
2254                            m = chain(0);
2255                            load_value(19, 2);      /* ether type */
2256                            load_const(0x6009);
2257                            emitop(OP_LE);
2258                            resolve_chain(m);
2259                    } else {
2260                            emitop(OP_OFFSET_ETHERTYPE);
2261                            load_value(0, 2);       /* ether type */
2262                            load_const(0x6000);
2263                            emitop(OP_GE);
2264                            emitop(OP_BRFL);
2265                            m = chain(0);
2266                            load_value(0, 2);       /* ether type */
2267                            load_const(0x6009);
2268                            emitop(OP_LE);
2269                            resolve_chain(m);
2270                            emitop(OP_OFFSET_POP);
2271                    }
2272                    opstack++;
2273                    next();
2274                    break;
2275            }

2277            if (EQ("vlan-id")) {
2278                    next();
2279                    if (tokentype != NUMBER)
2280                            pr_err("vlan id expected");
2281                    emitop(OP_OFFSET_ZERO);
2282                    ethertype_match(ETHERTYPE_VLAN);
2283                    emitop(OP_BRFL);
2284                    m = chain(0);
2285                    compare_value_mask(VLAN_ID_OFFSET, 2, tokenval,
2286                        VLAN_ID_MASK);
2287                    resolve_chain(m);
2288                    emitop(OP_OFFSET_POP);
2289                    opstack++;
2290                    next();
2291                    break;
2292            }

2294            if (EQ("apple")) {
2295                    /*
2296                     * Appletalk also appears in 802.2
2297                     * packets, so check for the ethertypes
2298                     * at offset 12 and 20 in the MAC header.
2299                     */
2300                    ethertype_match(ETHERTYPE_AT);
2301                    emitop(OP_BRTR);
2302                    m = chain(0);
2303                    ethertype_match(ETHERTYPE_AARP);
2304                    emitop(OP_BRTR);
```



```
2305                    m = chain(m);
2306                    compare_value(20, 2, ETHERTYPE_AT); /* 802.2 */
2307                    emitop(OP_BRTR);
2308                    m = chain(m);
2309                    compare_value(20, 2, ETHERTYPE_AARP); /* 802.2 */
2310                    resolve_chain(m);
2311                    opstack++;
2312                    next();
2313                    break;
2314            }

2316            if (EQ("vlan")) {
2317                    ethertype_match(ETHERTYPE_VLAN);
2318                    compare_value_mask(VLAN_ID_OFFSET, 2, 0, VLAN_ID_MASK);
2319                    emitop(OP_NOT);
2320                    emitop(OP_AND);
2321                    opstack++;
2322                    next();
2323                    break;
2324            }

2326            if (EQ("bootp") || EQ("dhcp")) {
2327                    ethertype_match(interface->network_type_ip);
2328                    emitop(OP_BRFL);
2329                    m = chain(0);
2330                    emitop(OP_OFFSET_LINK);
2331                    compare_value(9, 1, IPPROTO_UDP);
2332                    emitop(OP_OFFSET_POP);
2333                    emitop(OP_BRFL);
2334                    m = chain(m);
2335                    emitop(OP_OFFSET_IP);
2336                    compare_value(0, 4,
2337                        (IPPORT_BOOTPS << 16) | IPPORT_BOOTPC);
2338                    emitop(OP_BRTR);
2339                    m2 = chain(0);
2340                    compare_value(0, 4,
2341                        (IPPORT_BOOTPC << 16) | IPPORT_BOOTPS);
2342                    resolve_chain(m2);
2343                    emitop(OP_OFFSET_POP);
2344                    resolve_chain(m);
2345                    opstack++;
2346                    dir = ANY;
2347                    next();
2348                    break;
2349            }

2351            if (EQ("dhcp6")) {
2352                    ethertype_match(interface->network_type_ipv6);
2353                    emitop(OP_BRFL);
2354                    m = chain(0);
2355                    emitop(OP_OFFSET_LINK);
2356                    compare_value(6, 1, IPPROTO_UDP);
2357                    emitop(OP_OFFSET_POP);
2358                    emitop(OP_BRFL);
2359                    m = chain(m);
2360                    emitop(OP_OFFSET_IP);
2361                    compare_value(2, 2, IPPORT_DHCPV6S);
2362                    emitop(OP_BRTR);
2363                    m2 = chain(0);
2364                    compare_value(2, 2, IPPORT_DHCPV6C);
2365                    resolve_chain(m2);
2366                    emitop(OP_OFFSET_POP);
2367                    resolve_chain(m);
2368                    opstack++;
2369                    dir = ANY;
2370                    next();
```

```
2371                    break;
2372            }

2374            if (EQ("ethertype")) {
2375                    next();
2376                    if (tokentype != NUMBER)
2377                            pr_err("ether type expected");
2378                    ethertype_match(tokenval);
2379                    opstack++;
2380                    next();
2381                    break;
2382            }

2384            if (EQ("pppoe")) {
2385                    ethertype_match(ETHERTYPE_PPPOED);
2386                    ethertype_match(ETHERTYPE_PPPOES);
2387                    emitop(OP_OR);
2388                    opstack++;
2389                    next();
2390                    break;
2391            }

2393            if (EQ("inet")) {
2394                    next();
2395                    if (EQ("host"))
2396                            next();
2397                    if (tokentype != ALPHA && tokentype != ADDR_IP)
2398                            pr_err("host/IPv4 addr expected after inet");
2399                    ipaddr_match(dir, token, IPV4_ONLY);
2400                    opstack++;
2401                    next();
2402                    break;
2403            }

2405            if (EQ("inet6")) {
2406                    next();
2407                    if (EQ("host"))
2408                            next();
2409                    if (tokentype != ALPHA && tokentype != ADDR_IP6)
2410                            pr_err("host/IPv6 addr expected after inet6");
2411                    ipaddr_match(dir, token, IPV6_ONLY);
2412                    opstack++;
2413                    next();
2414                    break;
2415            }

2417            if (EQ("length")) {
2418                    emitop(OP_LOAD_LENGTH);
2419                    opstack++;
2420                    next();
2421                    break;
2422            }

2424            if (EQ("less")) {
2425                    next();
2426                    if (tokentype != NUMBER)
2427                            pr_err("packet length expected");
2428                    emitop(OP_LOAD_LENGTH);
2429                    load_const(tokenval);
2430                    emitop(OP_LT);
2431                    opstack++;
2432                    next();
2433                    break;
2434            }

2436            if (EQ("greater")) {
```



```
2437                    next();
2438                    if (tokentype != NUMBER)
2439                            pr_err("packet length expected");
2440                    emitop(OP_LOAD_LENGTH);
2441                    load_const(tokenval);
2442                    emitop(OP_GT);
2443                    opstack++;
2444                    next();
2445                    break;
2446            }

2448            if (EQ("nofrag")) {
2449                    emitop(OP_OFFSET_LINK);
2450                    compare_value_mask(6, 2, 0, 0x1fff);
2451                    emitop(OP_OFFSET_POP);
2452                    emitop(OP_BRFL);
2453                    m = chain(0);
2454                    ethertype_match(interface->network_type_ip);
2455                    resolve_chain(m);
2456                    opstack++;
2457                    next();
2458                    break;
2459            }

2461            if (EQ("net") || EQ("dstnet") || EQ("srcnet")) {
2462                    if (EQ("dstnet"))
2463                            dir = TO;
2464                    else if (EQ("srcnet"))
2465                            dir = FROM;
2466                    next();
2467                    netaddr_match(dir, token);
2468                    dir = ANY;
2469                    opstack++;
2470                    next();
2471                    break;
2472            }

2474            if (EQ("port") || EQ("srcport") || EQ("dstport")) {
2475                    if (EQ("dstport"))
2476                            dir = TO;
2477                    else if (EQ("srcport"))
2478                            dir = FROM;
2479                    next();
2480                    port_match(dir, token);
2481                    dir = ANY;
2482                    opstack++;
2483                    next();
2484                    break;
2485            }

2487            if (EQ("rpc")) {
2488                    uint_t vers, proc;
2489                    char savetoken[32];

2491                    vers = proc = -1;
2492                    next();
2493                    (void) strlcpy(savetoken, token, sizeof (savetoken));
2494                    next();
2495                    if (*token == ',') {
2496                            next();
2497                            if (tokentype != NUMBER)
2498                                    pr_err("version number expected");
2499                            vers = tokenval;
2500                            next();
2501                    }
2502                    if (*token == ',') {
```

```
2503                                next();
2504                                if (tokentype != NUMBER)
2505                                        pr_err("proc number expected");
2506                                proc = tokenval;
2507                                next();
2508                        }
2509                        rpc_match_prog(dir, savetoken, vers, proc);
2510                        dir = ANY;
2511                        opstack++;
2512                        break;
2513                }

2515                if (EQ("slp")) {
2516                        /* filter out TCP handshakes */
2517                        emitop(OP_OFFSET_LINK);
2518                        compare_value(9, 1, IPPROTO_TCP);
2519                        emitop(OP_LOAD_CONST);
2520                        emitval(52);
2521                        emitop(OP_LOAD_CONST);
2522                        emitval(2);
2523                        emitop(OP_LOAD_SHORT);
2524                        emitop(OP_GE);
2525                        emitop(OP_AND); /* proto == TCP && len < 52 */
2526                        emitop(OP_NOT);
2527                        emitop(OP_BRFL); /* pkt too short to be a SLP call */
2528                        m = chain(0);

2530                        emitop(OP_OFFSET_POP);
2531                        emitop(OP_OFFSET_SLP);
2532                        resolve_chain(m);
2533                        opstack++;
2534                        next();
2535                        break;
2536                }

2538                if (EQ("ldap")) {
2539                        dir = ANY;
2540                        port_match(dir, "ldap");
2541                        opstack++;
2542                        next();
2543                        break;
2544                }

2546                if (EQ("and") || EQ("or")) {
2547                        break;
2548                }

2550                if (EQ("zone")) {
2551                        next();
2552                        if (tokentype != NUMBER)
2553                                pr_err("zoneid expected");
2554                        zone_match(dir, BE_32((uint32_t)(tokenval)));
2555                        opstack++;
2556                        next();
2557                        break;
2558                }

2560                if (EQ("gateway")) {
2561                        next();
2562                        if (eaddr || tokentype != ALPHA)
2563                                pr_err("hostname required: %s", token);
2564                        etheraddr_match(dir, token);
2565                        dir = ANY;
2566                        emitop(OP_BRFL);
2567                        m = chain(0);
2568                        ipaddr_match(dir, token, IPV4_AND_IPV6);
```



```
2569                        emitop(OP_NOT);
2570                        resolve_chain(m);
2571                        opstack++;
2572                        next();
2573                }

2575                if (EQ("host") || EQ("between") ||
2576                    tokentype == ALPHA ||       /* assume its a hostname */
2577                    tokentype == ADDR_IP ||
2578                    tokentype == ADDR_IP6 ||
2579                    tokentype == ADDR_AT ||
2580                    tokentype == ADDR_ETHER) {
2581                        if (EQ("host") || EQ("between"))
2582                                next();
2583                        if (eaddr || tokentype == ADDR_ETHER) {
2584                                etheraddr_match(dir, token);
2585                        } else if (tokentype == ALPHA) {
2586                                ipaddr_match(dir, token, IPV4_AND_IPV6);
2587                        } else if (tokentype == ADDR_AT) {
2588                                ataddr_match(dir, token);
2589                        } else if (tokentype == ADDR_IP) {
2590                                ipaddr_match(dir, token, IPV4_ONLY);
2591                        } else {
2592                                ipaddr_match(dir, token, IPV6_ONLY);
2593                        }
2594                        dir = ANY;
2595                        eaddr = 0;
2596                        opstack++;
2597                        next();
2598                        break;
2599                }

2601                if (tokentype == NUMBER) {
2602                        load_const(tokenval);
2603                        opstack++;
2604                        next();
2605                        break;
2606                }

2608                break;  /* unknown token */
2609        }
2610 }

2612 struct optable {
2613         char *op_tok;
2614         enum optype op_type;
2615 };

2617 static struct optable
2618 mulops[] = {
2619         "*",    OP_MUL,
2620         "/",    OP_DIV,
2621         "%",    OP_REM,
2622         "&",    OP_AND,
2623         "",     OP_STOP,
2624 };

2626 static struct optable
2627 addops[] = {
2628         "+",    OP_ADD,
2629         "-",    OP_SUB,
2630         "|",    OP_OR,
2631         "^",    OP_XOR,
2632         "",     OP_STOP,
2633 };
```

```
2635 static struct optable
2636 compareops[] = {
2637         "==",   OP_EQ,
2638         "=",    OP_EQ,
2639         "!=",   OP_NE,
2640         ">",    OP_GT,
2641         ">=",   OP_GE,
2642         "<",    OP_LT,
2643         "<=",   OP_LE,
2644         "",     OP_STOP,
2645 };

2647 /*
2648  * Using the table, find the operator
2649  * that corresponds to the token.
2650  * Return 0 if not found.
2651  */
2652 static int
2653 find_op(char *tok, struct optable *table)
2654 {
2655         struct optable *op;

2657         for (op = table; *op->op_tok; op++) {
2658                 if (strcmp(tok, op->op_tok) == 0)
2659                         return (op->op_type);
2660         }

2662         return (0);
2663 }

2665 static void
2666 expr_mul()
2667 {
2668         int op;
2669         int s = opstack;

2671         primary();
2672         while (op = find_op(token, mulops)) {
2673                 next();
2674                 primary();
2675                 checkstack(s + 2);
2676                 emitop(op);
2677                 opstack--;
2678         }
2679 }

2681 static void
2682 expr_add()
2683 {
2684         int op, s = opstack;

2686         expr_mul();
2687         while (op = find_op(token, addops)) {
2688                 next();
2689                 expr_mul();
2690                 checkstack(s + 2);
2691                 emitop(op);
2692                 opstack--;
2693         }
2694 }

2696 static void
2697 expr_compare()
2698 {
2699         int op, s = opstack;
```



```
2701         expr_add();
2702         while (op = find_op(token, compareops)) {
2703                 next();
2704                 expr_add();
2705                 checkstack(s + 2);
2706                 emitop(op);
2707                 opstack--;
2708         }
2709 }

2711 /*
2712  * Alternation ("and") is difficult because
2713  * an implied "and" is acknowledge between
2714  * two adjacent primaries.  Just keep calling
2715  * the lower-level expression routine until
2716  * no value is added to the opstack.
2717  */
2718 static void
2719 alternation()
2720 {
2721         int m = 0;
2722         int s = opstack;

2724         expr_compare();
2725         checkstack(s + 1);
2726         for (;;) {
2727                 if (EQ("and"))
2728                         next();
2729                 emitop(OP_BRFL);
2730                 m = chain(m);
2731                 expr_compare();
2732                 if (opstack != s + 2)
2733                         break;
2734                 opstack--;
2735         }
2736         unemit(2);
2737         resolve_chain(m);
2738 }

2740 static void
2741 expression()
2742 {
2743         int m = 0;
2744         int s = opstack;

2746         alternation();
2747         while (EQ("or") || EQ(",")) {
2748                 emitop(OP_BRTR);
2749                 m = chain(m);
2750                 next();
2751                 alternation();
2752                 checkstack(s + 2);
2753                 opstack--;
2754         }
2755         resolve_chain(m);
2756 }

2758 /*
2759  * Take n args from the argv list
2760  * and concatenate them into a single string.
2761  */
2762 char *
2763 concat_args(char **argv, int argc)
2764 {
2765         int i, len;
2766         char *str, *p;
```

```
2768         /* First add the lengths of all the strings */
2769         len = 0;
2770         for (i = 0; i < argc; i++)
2771                 len += strlen(argv[i]) + 1;

2773         /* allocate the big string */
2774         str = (char *)malloc(len);
2775         if (str == NULL)
2776                 pr_err("no mem");

2778         p = str;

2780         /*
2781          * Concat the strings into the big
2782          * string using a space as separator
2783          */
2784         for (i = 0; i < argc; i++) {
2785                 strcpy(p, argv[i]);
2786                 p += strlen(p);
2787                 *p++ = ' ';
2788         }
2789         *--p = '\0';

2791         return (str);
2792 }

2794 /*
2795  * Take the expression in the string "expr"
2796  * and compile it into the code array.
2797  * Print the generated code if the print
2798  * arg is set.
2799  */
2800 void
2801 compile(char *expr, int print)
2802 {
2803         expr = strdup(expr);
2804         if (expr == NULL)
2805                 pr_err("no mem");
2806         curr_op = oplist;
2807         tkp = expr;
2808         dir = ANY;

2810         next();
2811         if (tokentype != EOL)
2812                 expression();
2813         emitop(OP_STOP);
2814         if (tokentype != EOL)
2815                 pr_err("invalid expression");
2816         optimize(oplist);
2817         if (print)
2818                 codeprint();
2819 }

2821 /*
2822  * Lookup hostname in the arp cache.
2823  */
2824 boolean_t
2825 arp_for_ether(char *hostname, struct ether_addr *ep)
2826 {
2827         struct arpreq ar;
2828         struct hostent *hp;
2829         struct sockaddr_in *sin;
2830         int error_num;
2831         int s;
```



```
2833         memset(&ar, 0, sizeof (ar));
2834         sin = (struct sockaddr_in *)&ar.arp_pa;
2835         sin->sin_family = AF_INET;
2836         hp = getipnodebyname(hostname, AF_INET, 0, &error_num);
2837         if (hp == NULL) {
2838                 return (B_FALSE);
2839         }
2840         memcpy(&sin->sin_addr, hp->h_addr, sizeof (sin->sin_addr));
2841         s = socket(AF_INET, SOCK_DGRAM, 0);
2842         if (s < 0) {
2843                 return (B_FALSE);
2844         }
2845         if (ioctl(s, SIOCGARP, &ar) < 0) {
2846                 close(s);
2847                 return (B_FALSE);
2848         }
2849         close(s);
2850         memcpy(ep->ether_addr_octet, ar.arp_ha.sa_data, sizeof (*ep));
2851         return (B_TRUE);
2852 }
```

```
**********************************************************
   39365 Mon Jul  9 14:38:07 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ip.c
dccp: snoop, build system fixes
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>

#include <sys/stropts.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip6.h>
#include <netinet/ip_icmp.h>
#include <netinet/icmp6.h>
#include <netinet/if_ether.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <tsol/label.h>
#include <sys/tsol/tndb.h>
#include <sys/tsol/label_macro.h>

#include "snoop.h"


/*
 * IPv6 extension header masks.  These are used by the print_ipv6_extensions()
 * function to return information to the caller about which extension headers
 * were processed.  This can be useful if the caller wants to know if the
 * packet is an IPv6 fragment, for example.
 */
#define SNOOP_HOPOPTS   0x01U
#define SNOOP_ROUTING   0x02U
```



```
#define SNOOP_DSTOPTS   0x04U
#define SNOOP_FRAGMENT  0x08U
#define SNOOP_AH        0x10U
#define SNOOP_ESP       0x20U
#define SNOOP_IPV6      0x40U

static void prt_routing_hdr(int, const struct ip6_rthdr *);
static void prt_fragment_hdr(int, const struct ip6_frag *);
static void prt_hbh_options(int, const struct ip6_hbh *);
static void prt_dest_options(int, const struct ip6_dest *);
static void print_route(const uchar_t *);
static void print_ipoptions(const uchar_t *, int);
static void print_ripso(const uchar_t *);
static void print_cipso(const uchar_t *);

/* Keep track of how many nested IP headers we have. */
unsigned int encap_levels;
unsigned int total_encap_levels = 1;

int
interpret_ip(int flags, const struct ip *ip, int fraglen)
{
        uchar_t *data;
        char buff[24];
        boolean_t isfrag = B_FALSE;
        boolean_t morefrag;
        uint16_t fragoffset;
        int hdrlen;
        uint16_t iplen, uitmp;

        if (ip->ip_v == IPV6_VERSION) {
                iplen = interpret_ipv6(flags, (ip6_t *)ip, fraglen);
                return (iplen);
        }

        if (encap_levels == 0)
                total_encap_levels = 0;
        encap_levels++;
        total_encap_levels++;

        hdrlen = ip->ip_hl * 4;
        data = ((uchar_t *)ip) + hdrlen;
        iplen = ntohs(ip->ip_len) - hdrlen;
        fraglen -= hdrlen;
        if (fraglen > iplen)
                fraglen = iplen;
        if (fraglen < 0) {
                (void) snprintf(get_sum_line(), MAXLINE,
                    "IP truncated: header missing %d bytes", -fraglen);
                encap_levels--;
                return (fraglen + iplen);
        }
        /*
         * We flag this as a fragment if the more fragments bit is set, or
         * if the fragment offset is non-zero.
         */
        morefrag = (ntohs(ip->ip_off) & IP_MF) == 0 ? B_FALSE : B_TRUE;
        fragoffset = (ntohs(ip->ip_off) & 0x1FFF) * 8;
        if (morefrag || fragoffset != 0)
                isfrag = B_TRUE;

        src_name = addrtoname(AF_INET, &ip->ip_src);
        dst_name = addrtoname(AF_INET, &ip->ip_dst);

        if (flags & F_SUM) {
                if (isfrag) {
```

```
128                         (void) snprintf(get_sum_line(), MAXLINE,
129                             "%s IP fragment ID=%d Offset=%-4d MF=%d TOS=0x%x "
130                             "TTL=%d",
131                             getproto(ip->ip_p),
132                             ntohs(ip->ip_id),
133                             fragoffset,
134                             morefrag,
135                             ip->ip_tos,
136                             ip->ip_ttl);
137                 } else {
138                         (void) strlcpy(buff, inet_ntoa(ip->ip_dst),
139                             sizeof (buff));
140                         uitmp = ntohs(ip->ip_len);
141                         (void) snprintf(get_sum_line(), MAXLINE,
142                             "IP  D=%s S=%s LEN=%u%s, ID=%d, TOS=0x%x, TTL=%d",
143                             buff,
144                             inet_ntoa(ip->ip_src),
145                             uitmp,
146                             iplen > fraglen ? "?" : "",
147                             ntohs(ip->ip_id),
148                             ip->ip_tos,
149                             ip->ip_ttl);
150                 }
151         }

153         if (flags & F_DTAIL) {
154                 show_header("IP:   ", "IP Header", iplen);
155                 show_space();
156                 (void) snprintf(get_line(0, 0), get_line_remain(),
157                     "Version = %d", ip->ip_v);
158                 (void) snprintf(get_line(0, 0), get_line_remain(),
159                     "Header length = %d bytes", hdrlen);
160                 (void) snprintf(get_line(0, 0), get_line_remain(),
161                     "Type of service = 0x%02x", ip->ip_tos);
162                 (void) snprintf(get_line(0, 0), get_line_remain(),
163                     "      xxx. .... = %d (precedence)",
164                     ip->ip_tos >> 5);
165                 (void) snprintf(get_line(0, 0), get_line_remain(),
166                     "      %s", getflag(ip->ip_tos, IPTOS_LOWDELAY,
167                     "low delay", "normal delay"));
168                 (void) snprintf(get_line(0, 0), get_line_remain(), "      %s",
169                     getflag(ip->ip_tos, IPTOS_THROUGHPUT,
170                     "high throughput", "normal throughput"));
171                 (void) snprintf(get_line(0, 0), get_line_remain(), "      %s",
172                     getflag(ip->ip_tos, IPTOS_RELIABILITY,
173                     "high reliability", "normal reliability"));
174                 (void) snprintf(get_line(0, 0), get_line_remain(), "      %s",
175                     getflag(ip->ip_tos, IPTOS_ECT,
176                     "ECN capable transport", "not ECN capable transport"));
177                 (void) snprintf(get_line(0, 0), get_line_remain(), "      %s",
178                     getflag(ip->ip_tos, IPTOS_CE,
179                     "ECN congestion experienced",
180                     "no ECN congestion experienced"));
181                 /* warning: ip_len is signed in netinet/ip.h */
182                 uitmp = ntohs(ip->ip_len);
183                 (void) snprintf(get_line(0, 0), get_line_remain(),
184                     "Total length = %u bytes%s", uitmp,
185                     iplen > fraglen ? " -- truncated" : "");
186                 (void) snprintf(get_line(0, 0), get_line_remain(),
187                     "Identification = %d", ntohs(ip->ip_id));
188                 /* warning: ip_off is signed in netinet/ip.h */
189                 uitmp = ntohs(ip->ip_off);
190                 (void) snprintf(get_line(0, 0), get_line_remain(),
191                     "Flags = 0x%x", uitmp >> 12);
192                 (void) snprintf(get_line(0, 0), get_line_remain(), "      %s",
193                     getflag(uitmp >> 8, IP_DF >> 8,
```



```
194                     "do not fragment", "may fragment"));
195                 (void) snprintf(get_line(0, 0), get_line_remain(), "      %s",
196                     getflag(uitmp >> 8, IP_MF >> 8,
197                     "more fragments", "last fragment"));
198                 (void) snprintf(get_line(0, 0), get_line_remain(),
199                     "Fragment offset = %u bytes",
200                     fragoffset);
201                 (void) snprintf(get_line(0, 0), get_line_remain(),
202                     "Time to live = %d seconds/hops",
203                     ip->ip_ttl);
204                 (void) snprintf(get_line(0, 0), get_line_remain(),
205                     "Protocol = %d (%s)", ip->ip_p,
206                     getproto(ip->ip_p));
207                 /*
208                  * XXX need to compute checksum and print whether it's correct
209                  */
210                 (void) snprintf(get_line(0, 0), get_line_remain(),
211                     "Header checksum = %04x",
212                     ntohs(ip->ip_sum));
213                 (void) snprintf(get_line(0, 0), get_line_remain(),
214                     "Source address = %s, %s",
215                     inet_ntoa(ip->ip_src), addrtoname(AF_INET, &ip->ip_src));
216                 (void) snprintf(get_line(0, 0), get_line_remain(),
217                     "Destination address = %s, %s",
218                     inet_ntoa(ip->ip_dst), addrtoname(AF_INET, &ip->ip_dst));

220                 /* Print IP options - if any */

222                 print_ipoptions((const uchar_t *)(ip + 1),
223                     hdrlen - sizeof (struct ip));
224                 show_space();
225         }

227         /*
228          * If we are in detail mode, and this is not the first fragment of
229          * a fragmented packet, print out a little line stating this.
230          * Otherwise, go to the next protocol layer only if this is not a
231          * fragment, or we are in detail mode and this is the first fragment
232          * of a fragmented packet.
233          */
234         if (flags & F_DTAIL && fragoffset != 0) {
235                 (void) snprintf(get_detail_line(0, 0), MAXLINE,
236                     "%s:  [%d byte(s) of data, continuation of IP ident=%d]",
237                     getproto(ip->ip_p),
238                     iplen,
239                     ntohs(ip->ip_id));
240         } else if (!isfrag || (flags & F_DTAIL) && isfrag && fragoffset == 0) {
241                 /* go to the next protocol layer */

243                 if (fraglen > 0) {
244                         switch (ip->ip_p) {
245                         case IPPROTO_IP:
246                                 break;
247                         case IPPROTO_ENCAP:
248                                 (void) interpret_ip(flags,
249                                     /* LINTED: alignment */
250                                     (const struct ip *)data, fraglen);
251                                 break;
252                         case IPPROTO_ICMP:
253                                 (void) interpret_icmp(flags,
254                                     /* LINTED: alignment */
255                                     (struct icmp *)data, iplen, fraglen);
256                                 break;
257                         case IPPROTO_IGMP:
258                                 interpret_igmp(flags, data, iplen, fraglen);
259                                 break;
```

```
260                         case IPPROTO_GGP:
261                                 break;
262                         case IPPROTO_TCP:
263                                 (void) interpret_tcp(flags,
264                                     (struct tcphdr *)data, iplen, fraglen);
265                                 break;

267                         case IPPROTO_ESP:
268                                 (void) interpret_esp(flags, data, iplen,
269                                     fraglen);
270                                 break;
271                         case IPPROTO_AH:
272                                 (void) interpret_ah(flags, data, iplen,
273                                     fraglen);
274                                 break;

276                         case IPPROTO_OSPF:
277                                 interpret_ospf(flags, data, iplen, fraglen);
278                                 break;

280                         case IPPROTO_EGP:
281                         case IPPROTO_PUP:
282                                 break;
283                         case IPPROTO_UDP:
284                                 (void) interpret_udp(flags,
285                                     (struct udphdr *)data, iplen, fraglen);
286                                 break;

288                         case IPPROTO_IDP:
289                         case IPPROTO_HELLO:
290                         case IPPROTO_ND:
291                         case IPPROTO_RAW:
292                                 break;
293                         case IPPROTO_IPV6:      /* IPV6 encap */
294                                 /* LINTED: alignment */
295                                 (void) interpret_ipv6(flags, (ip6_t *)data,
296                                     iplen);
297                                 break;
298                         case IPPROTO_SCTP:
299                                 (void) interpret_sctp(flags,
300                                     (struct sctp_hdr *)data, iplen, fraglen);
301                                 break;
302                         case IPPROTO_DCCP:
303                                 (void) interpret_dccp(flags,
304                                     (struct dccphdr *)data, iplen, fraglen);
305                                 break;
306 #endif /* ! codereview */
307                         }
308                 }
309         }

311         encap_levels--;
312         return (iplen);
313 }

315 int
316 interpret_ipv6(int flags, const ip6_t *ip6h, int fraglen)
317 {
318         uint8_t *data;
319         int hdrlen, iplen;
320         int version, flow, class;
321         uchar_t proto;
322         boolean_t isfrag = B_FALSE;
323         uint8_t extmask;
324         /*
325          * The print_srcname and print_dstname strings are the hostname
```



```
326          * parts of the verbose IPv6 header output, including the comma
327          * and the space after the litteral address strings.
328          */
329         char print_srcname[MAXHOSTNAMELEN + 2];
330         char print_dstname[MAXHOSTNAMELEN + 2];
331         char src_addrstr[INET6_ADDRSTRLEN];
332         char dst_addrstr[INET6_ADDRSTRLEN];

334         iplen = ntohs(ip6h->ip6_plen);
335         hdrlen = IPV6_HDR_LEN;
336         fraglen -= hdrlen;
337         if (fraglen < 0)
338                 return (fraglen + hdrlen);
339         data = ((uint8_t *)ip6h) + hdrlen;

341         proto = ip6h->ip6_nxt;

343         src_name = addrtoname(AF_INET6, &ip6h->ip6_src);
344         dst_name = addrtoname(AF_INET6, &ip6h->ip6_dst);

346         /*
347          * Use endian-aware masks to extract traffic class and
348          * flowinfo.  Also, flowinfo is now 20 bits and class 8
349          * rather than 24 and 4.
350          */
351         class = ntohl((ip6h->ip6_vcf & IPV6_FLOWINFO_TCLASS) >> 20);
352         flow = ntohl(ip6h->ip6_vcf & IPV6_FLOWINFO_FLOWLABEL);

354         /*
355          * NOTE: the F_SUM and F_DTAIL flags are mutually exclusive,
356          * so the code within the first part of the following if statement
357          * will not affect the detailed printing of the packet.
358          */
359         if (flags & F_SUM) {
360                 (void) snprintf(get_sum_line(), MAXLINE,
361                     "IPv6  S=%s D=%s LEN=%d HOPS=%d CLASS=0x%x FLOW=0x%x",
362                     src_name, dst_name, iplen, ip6h->ip6_hops, class, flow);
363         } else if (flags & F_DTAIL) {

365                 (void) inet_ntop(AF_INET6, &ip6h->ip6_src, src_addrstr,
366                     INET6_ADDRSTRLEN);
367                 (void) inet_ntop(AF_INET6, &ip6h->ip6_dst, dst_addrstr,
368                     INET6_ADDRSTRLEN);

370                 version = ntohl(ip6h->ip6_vcf) >> 28;

372                 if (strcmp(src_name, src_addrstr) == 0) {
373                         print_srcname[0] = '\0';
374                 } else {
375                         snprintf(print_srcname, sizeof (print_srcname),
376                             ", %s", src_name);
377                 }

379                 if (strcmp(dst_name, dst_addrstr) == 0) {
380                         print_dstname[0] = '\0';
381                 } else {
382                         snprintf(print_dstname, sizeof (print_dstname),
383                             ", %s", dst_name);
384                 }

386                 show_header("IPv6:   ", "IPv6 Header", iplen);
387                 show_space();

389                 (void) snprintf(get_line(0, 0), get_line_remain(),
390                     "Version = %d", version);
391                 (void) snprintf(get_line(0, 0), get_line_remain(),
```

```
392                     "Traffic Class = %d", class);
393             (void) snprintf(get_line(0, 0), get_line_remain(),
394                 "Flow label = 0x%x", flow);
395             (void) snprintf(get_line(0, 0), get_line_remain(),
396                 "Payload length = %d", iplen);
397             (void) snprintf(get_line(0, 0), get_line_remain(),
398                 "Next Header = %d (%s)", proto,
399                 getproto(proto));
400             (void) snprintf(get_line(0, 0), get_line_remain(),
401                 "Hop Limit = %d", ip6h->ip6_hops);
402             (void) snprintf(get_line(0, 0), get_line_remain(),
403                 "Source address = %s%s", src_addrstr, print_srcname);
404             (void) snprintf(get_line(0, 0), get_line_remain(),
405                 "Destination address = %s%s", dst_addrstr, print_dstname);

407             show_space();
408         }

410         /*
411          * Print IPv6 Extension Headers, or skip them in the summary case.
412          * Set isfrag to true if one of the extension headers encounterred
413          * was a fragment header.
414          */
415         if (proto == IPPROTO_HOPOPTS || proto == IPPROTO_DSTOPTS ||
416             proto == IPPROTO_ROUTING || proto == IPPROTO_FRAGMENT) {
417             extmask = print_ipv6_extensions(flags, &data, &proto, &iplen,
418                 &fraglen);
419             if ((extmask & SNOOP_FRAGMENT) != 0) {
420                     isfrag = B_TRUE;
421             }
422         }

424         /*
425          * We only want to print upper layer information if this is not
426          * a fragment, or if we're printing in detail.  Note that the
427          * proto variable will be set to IPPROTO_NONE if this is a fragment
428          * with a non-zero fragment offset.
429          */
430         if (!isfrag || flags & F_DTAIL) {
431                 /* go to the next protocol layer */

433                 switch (proto) {
434                 case IPPROTO_IP:
435                         break;
436                 case IPPROTO_ENCAP:
437                         /* LINTED: alignment */
438                         (void) interpret_ip(flags, (const struct ip *)data,
439                             fraglen);
440                         break;
441                 case IPPROTO_ICMPV6:
442                         /* LINTED: alignment */
443                         (void) interpret_icmpv6(flags, (icmp6_t *)data, iplen,
444                             fraglen);
445                         break;
446                 case IPPROTO_IGMP:
447                         interpret_igmp(flags, data, iplen, fraglen);
448                         break;
449                 case IPPROTO_GGP:
450                         break;
451                 case IPPROTO_TCP:
452                         (void) interpret_tcp(flags, (struct tcphdr *)data,
453                             iplen, fraglen);
454                         break;
455                 case IPPROTO_ESP:
456                         (void) interpret_esp(flags, data, iplen, fraglen);
457                         break;
```



```
458                 case IPPROTO_AH:
459                         (void) interpret_ah(flags, data, iplen, fraglen);
460                         break;
461                 case IPPROTO_EGP:
462                 case IPPROTO_PUP:
463                         break;
464                 case IPPROTO_UDP:
465                         (void) interpret_udp(flags, (struct udphdr *)data,
466                             iplen, fraglen);
467                         break;
468                 case IPPROTO_IDP:
469                 case IPPROTO_HELLO:
470                 case IPPROTO_ND:
471                 case IPPROTO_RAW:
472                         break;
473                 case IPPROTO_IPV6:
474                         /* LINTED: alignment */
475                         (void) interpret_ipv6(flags, (const ip6_t *)data,
476                             iplen);
477                         break;
478                 case IPPROTO_SCTP:
479                         (void) interpret_sctp(flags, (struct sctp_hdr *)data,
480                             iplen, fraglen);
481                         break;
482                 case IPPROTO_OSPF:
483                         interpret_ospf6(flags, data, iplen, fraglen);
484                         break;
485                 case IPPROTO_DCCP:
486                         (void) interpret_dccp(flags, (struct dccphdr *)data,
487                             iplen, fraglen);
488                         break;
489 #endif /* ! codereview */
490                 }
491         }

493         return (iplen);
494 }

496 /*
497  * ip_ext: data including the extension header.
498  * iplen: length of the data remaining in the packet.
499  * Returns a mask of IPv6 extension headers it processed.
500  */
501 uint8_t
502 print_ipv6_extensions(int flags, uint8_t **hdr, uint8_t *next, int *iplen,
503     int *fraglen)
504 {
505         uint8_t *data_ptr;
506         uchar_t proto = *next;
507         boolean_t is_extension_header;
508         struct ip6_hbh *ipv6ext_hbh;
509         struct ip6_dest *ipv6ext_dest;
510         struct ip6_rthdr *ipv6ext_rthdr;
511         struct ip6_frag *ipv6ext_frag;
512         uint32_t exthdrlen;
513         uint8_t extmask = 0;

515         if ((hdr == NULL) || (*hdr == NULL) || (next == NULL) || (iplen == 0))
516                 return (0);

518         data_ptr = *hdr;
519         is_extension_header = B_TRUE;
520         while (is_extension_header) {

522                 /*
523                  * There must be at least enough data left to read the
```

```
524                  * next header and header length fields from the next
525                  * header.
526                  */
527                 if (*fraglen < 2) {
528                         return (extmask);
529                 }

531                 switch (proto) {
532                 case IPPROTO_HOPOPTS:
533                         ipv6ext_hbh = (struct ip6_hbh *)data_ptr;
534                         exthdrlen = 8 + ipv6ext_hbh->ip6h_len * 8;
535                         if (*fraglen <= exthdrlen) {
536                                 return (extmask);
537                         }
538                         prt_hbh_options(flags, ipv6ext_hbh);
539                         extmask |= SNOOP_HOPOPTS;
540                         proto = ipv6ext_hbh->ip6h_nxt;
541                         break;
542                 case IPPROTO_DSTOPTS:
543                         ipv6ext_dest = (struct ip6_dest *)data_ptr;
544                         exthdrlen = 8 + ipv6ext_dest->ip6d_len * 8;
545                         if (*fraglen <= exthdrlen) {
546                                 return (extmask);
547                         }
548                         prt_dest_options(flags, ipv6ext_dest);
549                         extmask |= SNOOP_DSTOPTS;
550                         proto = ipv6ext_dest->ip6d_nxt;
551                         break;
552                 case IPPROTO_ROUTING:
553                         ipv6ext_rthdr = (struct ip6_rthdr *)data_ptr;
554                         exthdrlen = 8 + ipv6ext_rthdr->ip6r_len * 8;
555                         if (*fraglen <= exthdrlen) {
556                                 return (extmask);
557                         }
558                         prt_routing_hdr(flags, ipv6ext_rthdr);
559                         extmask |= SNOOP_ROUTING;
560                         proto = ipv6ext_rthdr->ip6r_nxt;
561                         break;
562                 case IPPROTO_FRAGMENT:
563                         /* LINTED: alignment */
564                         ipv6ext_frag = (struct ip6_frag *)data_ptr;
565                         exthdrlen = sizeof (struct ip6_frag);
566                         if (*fraglen <= exthdrlen) {
567                                 return (extmask);
568                         }
569                         prt_fragment_hdr(flags, ipv6ext_frag);
570                         extmask |= SNOOP_FRAGMENT;
571                         /*
572                          * If this is not the first fragment, forget about
573                          * the rest of the packet, snoop decoding is
574                          * stateless.
575                          */
576                         if ((ipv6ext_frag->ip6f_offlg & IP6F_OFF_MASK) != 0)
577                                 proto = IPPROTO_NONE;
578                         else
579                                 proto = ipv6ext_frag->ip6f_nxt;
580                         break;
581                 default:
582                         is_extension_header = B_FALSE;
583                         break;
584                 }

586                 if (is_extension_header) {
587                         *iplen -= exthdrlen;
588                         *fraglen -= exthdrlen;
589                         data_ptr += exthdrlen;
```



```
590                 }
591         }

593         *next = proto;
594         *hdr = data_ptr;
595         return (extmask);
596 }

598 static void
599 print_ipoptions(const uchar_t *opt, int optlen)
600 {
601         int len;
602         int remain;
603         char *line;
604         const char *truncstr;

606         if (optlen <= 0) {
607                 (void) snprintf(get_line(0, 0), get_line_remain(),
608                     "No options");
609                 return;
610         }

612         (void) snprintf(get_line(0, 0), get_line_remain(),
613             "Options: (%d bytes)", optlen);

615         while (optlen > 0) {
616                 line = get_line(0, 0);
617                 remain = get_line_remain();
618                 len = opt[1];
619                 truncstr = len > optlen ? "?" : "";
620                 switch (opt[0]) {
621                 case IPOPT_EOL:
622                         (void) strlcpy(line, "  - End of option list", remain);
623                         return;
624                 case IPOPT_NOP:
625                         (void) strlcpy(line, "  - No op", remain);
626                         len = 1;
627                         break;
628                 case IPOPT_RR:
629                         (void) snprintf(line, remain,
630                             "  - Record route (%d bytes%s)", len, truncstr);
631                         print_route(opt);
632                         break;
633                 case IPOPT_TS:
634                         (void) snprintf(line, remain,
635                             "  - Time stamp (%d bytes%s)", len, truncstr);
636                         break;
637                 case IPOPT_SECURITY:
638                         (void) snprintf(line, remain, "  - RIPSO (%d bytes%s)",
639                             len, truncstr);
640                         print_ripso(opt);
641                         break;
642                 case IPOPT_COMSEC:
643                         (void) snprintf(line, remain, "  - CIPSO (%d bytes%s)",
644                             len, truncstr);
645                         print_cipso(opt);
646                         break;
647                 case IPOPT_LSRR:
648                         (void) snprintf(line, remain,
649                             "  - Loose source route (%d bytes%s)", len,
650                             truncstr);
651                         print_route(opt);
652                         break;
653                 case IPOPT_SATID:
654                         (void) snprintf(line, remain,
655                             "  - SATNET Stream id (%d bytes%s)",
```

```
656                             len, truncstr);
657                     break;
658             case IPOPT_SSRR:
659                     (void) snprintf(line, remain,
660                         "  - Strict source route, (%d bytes%s)", len,
661                         truncstr);
662                     print_route(opt);
663                     break;
664             default:
665                     (void) snprintf(line, remain,
666                         "  - Option %d (unknown - %d bytes%s) %s",
667                         opt[0], len, truncstr,
668                         tohex((char *)&opt[2], len - 2));
669                     break;
670             }
671             if (len <= 0) {
672                     (void) snprintf(line, remain,
673                         "  - Incomplete option len %d", len);
674                     break;
675             }
676             opt += len;
677             optlen -= len;
678     }
679 }

681 static void
682 print_route(const uchar_t *opt)
683 {
684     int len, pointer, remain;
685     struct in_addr addr;
686     char *line;

688     len = opt[1];
689     pointer = opt[2];

691     (void) snprintf(get_line(0, 0), get_line_remain(),
692         "    Pointer = %d", pointer);

694     pointer -= IPOPT_MINOFF;
695     opt += (IPOPT_OFFSET + 1);
696     len -= (IPOPT_OFFSET + 1);

698     while (len > 0) {
699             line = get_line(0, 0);
700             remain = get_line_remain();
701             memcpy((char *)&addr, opt, sizeof (addr));
702             if (addr.s_addr == INADDR_ANY)
703                     (void) strlcpy(line, "      -", remain);
704             else
705                     (void) snprintf(line, remain, "      %s",
706                         addrtoname(AF_INET, &addr));
707             if (pointer == 0)
708                     (void) strlcat(line, "  <-- (current)", remain);

710             opt += sizeof (addr);
711             len -= sizeof (addr);
712             pointer -= sizeof (addr);
713     }
714 }

716 char *
717 getproto(int p)
718 {
719     switch (p) {
720     case IPPROTO_HOPOPTS:   return ("IPv6-HopOpts");
721     case IPPROTO_IPV6:      return ("IPv6");
```



```
722     case IPPROTO_ROUTING:   return ("IPv6-Route");
723     case IPPROTO_FRAGMENT:  return ("IPv6-Frag");
724     case IPPROTO_RSVP:      return ("RSVP");
725     case IPPROTO_ENCAP:     return ("IP-in-IP");
726     case IPPROTO_AH:        return ("AH");
727     case IPPROTO_ESP:       return ("ESP");
728     case IPPROTO_ICMP:      return ("ICMP");
729     case IPPROTO_ICMPV6:    return ("ICMPv6");
730     case IPPROTO_DSTOPTS:   return ("IPv6-DstOpts");
731     case IPPROTO_IGMP:      return ("IGMP");
732     case IPPROTO_GGP:       return ("GGP");
733     case IPPROTO_TCP:       return ("TCP");
734     case IPPROTO_EGP:       return ("EGP");
735     case IPPROTO_PUP:       return ("PUP");
736     case IPPROTO_UDP:       return ("UDP");
737     case IPPROTO_IDP:       return ("IDP");
738     case IPPROTO_HELLO:     return ("HELLO");
739     case IPPROTO_ND:        return ("ND");
740     case IPPROTO_EON:       return ("EON");
741     case IPPROTO_RAW:       return ("RAW");
742     case IPPROTO_OSPF:      return ("OSPF");
743     default:                return ("");
744     }
745 }

747 static void
748 prt_routing_hdr(int flags, const struct ip6_rthdr *ipv6ext_rthdr)
749 {
750     uint8_t nxt_hdr;
751     uint8_t type;
752     uint32_t len;
753     uint8_t segleft;
754     uint32_t numaddrs;
755     int i;
756     struct ip6_rthdr0 *ipv6ext_rthdr0;
757     struct in6_addr *addrs;
758     char addr[INET6_ADDRSTRLEN];

760     /* in summary mode, we don't do anything. */
761     if (flags & F_SUM) {
762             return;
763     }

765     nxt_hdr = ipv6ext_rthdr->ip6r_nxt;
766     type = ipv6ext_rthdr->ip6r_type;
767     len = 8 * (ipv6ext_rthdr->ip6r_len + 1);
768     segleft = ipv6ext_rthdr->ip6r_segleft;

770     show_header("IPv6-Route:  ", "IPv6 Routing Header", 0);
771     show_space();

773     (void) snprintf(get_line(0, 0), get_line_remain(),
774         "Next header = %d (%s)", nxt_hdr, getproto(nxt_hdr));
775     (void) snprintf(get_line(0, 0), get_line_remain(),
776         "Header length = %d", len);
777     (void) snprintf(get_line(0, 0), get_line_remain(),
778         "Routing type = %d", type);
779     (void) snprintf(get_line(0, 0), get_line_remain(),
780         "Segments left = %d", segleft);

782     if (type == IPV6_RTHDR_TYPE_0) {
783             /*
784              * XXX This loop will print all addresses in the routing header,
785              * XXX not just the segments left.
786              * XXX (The header length field is twice the number of
787              * XXX addresses)
```

```
788                  * XXX At some future time, we may want to change this
789                  * XXX to differentiate between the hops yet to do
790                  * XXX and the hops already taken.
791                  */
792                 /* LINTED: alignment */
793                 ipv6ext_rthdr0 = (struct ip6_rthdr0 *)ipv6ext_rthdr;
794                 numaddrs = ipv6ext_rthdr0->ip6r0_len / 2;
795                 addrs = (struct in6_addr *)(ipv6ext_rthdr0 + 1);
796                 for (i = 0; i < numaddrs; i++) {
797                         (void) inet_ntop(AF_INET6, &addrs[i], addr,
798                             INET6_ADDRSTRLEN);
799                         (void) snprintf(get_line(0, 0), get_line_remain(),
800                             "address[%d]=%s", i, addr);
801                 }
802         }

804         show_space();
805 }

807 static void
808 prt_fragment_hdr(int flags, const struct ip6_frag *ipv6ext_frag)
809 {
810         boolean_t morefrag;
811         uint16_t fragoffset;
812         uint8_t nxt_hdr;
813         uint32_t fragident;

815         /* extract the various fields from the fragment header */
816         nxt_hdr = ipv6ext_frag->ip6f_nxt;
817         morefrag = (ipv6ext_frag->ip6f_offlg & IP6F_MORE_FRAG) == 0
818             ? B_FALSE : B_TRUE;
819         fragoffset = ntohs(ipv6ext_frag->ip6f_offlg & IP6F_OFF_MASK);
820         fragident = ntohl(ipv6ext_frag->ip6f_ident);

822         if (flags & F_SUM) {
823                 (void) snprintf(get_sum_line(), MAXLINE,
824                     "IPv6 fragment ID=%u Offset=%-4d MF=%d",
825                     fragident,
826                     fragoffset,
827                     morefrag);
828         } else { /* F_DTAIL */
829                 show_header("IPv6-Frag:  ", "IPv6 Fragment Header", 0);
830                 show_space();

832                 (void) snprintf(get_line(0, 0), get_line_remain(),
833                     "Next Header = %d (%s)", nxt_hdr, getproto(nxt_hdr));
834                 (void) snprintf(get_line(0, 0), get_line_remain(),
835                     "Fragment Offset = %d", fragoffset);
836                 (void) snprintf(get_line(0, 0), get_line_remain(),
837                     "More Fragments Flag = %s", morefrag ? "true" : "false");
838                 (void) snprintf(get_line(0, 0), get_line_remain(),
839                     "Identification = %u", fragident);

841                 show_space();
842         }
843 }

845 static void
846 print_ip6opt_ls(const uchar_t *data, unsigned int op_len)
847 {
848         uint32_t doi;
849         uint8_t sotype, solen;
850         uint16_t value, value2;
851         char *cp;
852         int remlen;
853         boolean_t printed;
```



```
855         (void) snprintf(get_line(0, 0), get_line_remain(),
856             "Labeled Security Option len = %u bytes%s", op_len,
857             op_len < sizeof (uint32_t) || (op_len & 1) != 0 ? "?" : "");
858         if (op_len < sizeof (uint32_t))
859                 return;
860         GETINT32(doi, data);
861         (void) snprintf(get_line(0, 0), get_line_remain(),
862             "    DOI = %d (%s)", doi, doi == IP6LS_DOI_V4 ? "IPv4" : "???");
863         op_len -= sizeof (uint32_t);
864         while (op_len > 0) {
865                 GETINT8(sotype, data);
866                 if (op_len < 2) {
867                         (void) snprintf(get_line(0, 0), get_line_remain(),
868                             "    truncated %u suboption (no len)", sotype);
869                         break;
870                 }
871                 GETINT8(solen, data);
872                 if (solen < 2 || solen > op_len) {
873                         (void) snprintf(get_line(0, 0), get_line_remain(),
874                             "    bad %u suboption (len 2 <= %u <= %u)",
875                             sotype, solen, op_len);
876                         if (solen < 2)
877                                 solen = 2;
878                         if (solen > op_len)
879                                 solen = op_len;
880                 }
881                 op_len -= solen;
882                 solen -= 2;
883                 cp = get_line(0, 0);
884                 remlen = get_line_remain();
885                 (void) strlcpy(cp, "    ", remlen);
886                 cp += 4;
887                 remlen -= 4;
888                 printed = B_TRUE;
889                 switch (sotype) {
890                 case IP6LS_TT_LEVEL:
891                         if (solen != 2) {
892                                 printed = B_FALSE;
893                                 break;
894                         }
895                         GETINT16(value, data);
896                         (void) snprintf(cp, remlen, "Level %u", value);
897                         solen = 0;
898                         break;
899                 case IP6LS_TT_VECTOR:
900                         (void) strlcpy(cp, "Bit-Vector: ", remlen);
901                         remlen -= strlen(cp);
902                         cp += strlen(cp);
903                         while (solen > 1) {
904                                 GETINT16(value, data);
905                                 solen -= 2;
906                                 (void) snprintf(cp, remlen, "%04x", value);
907                                 remlen -= strlen(cp);
908                                 cp += strlen(cp);
909                         }
910                         break;
911                 case IP6LS_TT_ENUM:
912                         (void) strlcpy(cp, "Enumeration:", remlen);
913                         remlen -= strlen(cp);
914                         cp += strlen(cp);
915                         while (solen > 1) {
916                                 GETINT16(value, data);
917                                 solen -= 2;
918                                 (void) snprintf(cp, remlen, " %u", value);
919                                 remlen -= strlen(cp);
```

```
920                                 cp += strlen(cp);
921                         }
922                         break;
923                 case IP6LS_TT_RANGES:
924                         (void) strlcpy(cp, "Ranges:", remlen);
925                         remlen -= strlen(cp);
926                         cp += strlen(cp);
927                         while (solen > 3) {
928                                 GETINT16(value, data);
929                                 GETINT16(value2, data);
930                                 solen -= 4;
931                                 (void) snprintf(cp, remlen, " %u-%u", value,
932                                     value2);
933                                 remlen -= strlen(cp);
934                                 cp += strlen(cp);
935                         }
936                         break;
937                 case IP6LS_TT_V4:
938                         (void) strlcpy(cp, "IPv4 Option", remlen);
939                         print_ipoptions(data, solen);
940                         solen = 0;
941                         break;
942                 case IP6LS_TT_DEST:
943                         (void) snprintf(cp, remlen,
944                             "Destination-Only Data length %u", solen);
945                         solen = 0;
946                         break;
947                 default:
948                         (void) snprintf(cp, remlen,
949                             "   unknown %u suboption (len %u)", sotype, solen);
950                         solen = 0;
951                         break;
952                 }
953                 if (solen != 0) {
954                         if (printed) {
955                                 cp = get_line(0, 0);
956                                 remlen = get_line_remain();
957                         }
958                         (void) snprintf(cp, remlen,
959                             "    malformed %u suboption (remaining %u)",
960                             sotype, solen);
961                         data += solen;
962                 }
963         }
964 }

966 static void
967 prt_hbh_options(int flags, const struct ip6_hbh *ipv6ext_hbh)
968 {
969         const uint8_t *data, *ndata;
970         uint32_t len;
971         uint8_t op_type;
972         uint8_t op_len;
973         uint8_t nxt_hdr;

975         /* in summary mode, we don't do anything. */
976         if (flags & F_SUM) {
977                 return;
978         }

980         show_header("IPv6-HopOpts:  ", "IPv6 Hop-by-Hop Options Header", 0);
981         show_space();

983         /*
984          * Store the lengh of this ext hdr in bytes.  The caller has
985          * ensured that there is at least len bytes of data left.
```



```
986          */
987         len = ipv6ext_hbh->ip6h_len * 8 + 8;

989         ndata = (const uint8_t *)ipv6ext_hbh + 2;
990         len -= 2;

992         nxt_hdr = ipv6ext_hbh->ip6h_nxt;
993         (void) snprintf(get_line(0, 0), get_line_remain(),
994             "Next Header = %u (%s)", nxt_hdr, getproto(nxt_hdr));

996         while (len > 0) {
997                 data = ndata;
998                 GETINT8(op_type, data);
999                 /* This is the only one-octet IPv6 option */
1000                if (op_type == IP6OPT_PAD1) {
1001                        (void) snprintf(get_line(0, 0), get_line_remain(),
1002                            "pad1 option ");
1003                        len--;
1004                        ndata = data;
1005                        continue;
1006                }
1007                GETINT8(op_len, data);
1008                if (len < 2 || op_len + 2 > len) {
1009                        (void) snprintf(get_line(0, 0), get_line_remain(),
1010                            "Error: option %u truncated (%u + 2 > %u)",
1011                            op_type, op_len, len);
1012                        op_len = len - 2;
1013                        /*
1014                         * Continue processing the malformed option so that we
1015                         * can display as much as possible.
1016                         */
1017                }

1019                /* advance pointers to the next option */
1020                len -= op_len + 2;
1021                ndata = data + op_len;

1023                /* process this option */
1024                switch (op_type) {
1025                case IP6OPT_PADN:
1026                        (void) snprintf(get_line(0, 0), get_line_remain(),
1027                            "padN option len = %u", op_len);
1028                        break;
1029                case IP6OPT_JUMBO: {
1030                        uint32_t payload_len;

1032                        (void) snprintf(get_line(0, 0), get_line_remain(),
1033                            "Jumbo Payload Option len = %u bytes%s", op_len,
1034                            op_len == sizeof (uint32_t) ? "" : "?");
1035                        if (op_len == sizeof (uint32_t)) {
1036                                GETINT32(payload_len, data);
1037                                (void) snprintf(get_line(0, 0),
1038                                    get_line_remain(),
1039                                    "Jumbo Payload Length = %u bytes",
1040                                    payload_len);
1041                        }
1042                        break;
1043                }
1044                case IP6OPT_ROUTER_ALERT: {
1045                        uint16_t value;
1046                        const char *label[] = {"MLD", "RSVP", "AN"};

1048                        (void) snprintf(get_line(0, 0), get_line_remain(),
1049                            "Router Alert Option len = %u bytes%s", op_len,
1050                            op_len == sizeof (uint16_t) ? "" : "?");
1051                        if (op_len == sizeof (uint16_t)) {
```

```
1052                                GETINT16(value, data);
1053                                (void) snprintf(get_line(0, 0),
1054                                    get_line_remain(),
1055                                    "Alert Type = %d (%s)", value,
1056                                    value < sizeof (label) / sizeof (label[0]) ?
1057                                    label[value] : "???");
1058                        }
1059                        break;
1060                }
1061                case IP6OPT_LS:
1062                        print_ip6opt_ls(data, op_len);
1063                        break;
1064                default:
1065                        (void) snprintf(get_line(0, 0), get_line_remain(),
1066                            "Option type = %u, len = %u", op_type, op_len);
1067                        break;
1068                }
1069        }

1071        show_space();
1072 }

1074 static void
1075 prt_dest_options(int flags, const struct ip6_dest *ipv6ext_dest)
1076 {
1077        const uint8_t *data, *ndata;
1078        uint32_t len;
1079        uint8_t op_type;
1080        uint32_t op_len;
1081        uint8_t nxt_hdr;
1082        uint8_t value;

1084        /* in summary mode, we don't do anything. */
1085        if (flags & F_SUM) {
1086                return;
1087        }

1089        show_header("IPv6-DstOpts:  ", "IPv6 Destination Options Header", 0);
1090        show_space();

1092        /*
1093         * Store the length of this ext hdr in bytes.  The caller has
1094         * ensured that there is at least len bytes of data left.
1095         */
1096        len = ipv6ext_dest->ip6d_len * 8 + 8;

1098        ndata = (const uint8_t *)ipv6ext_dest + 2; /* skip hdr/len */
1099        len -= 2;

1101        nxt_hdr = ipv6ext_dest->ip6d_nxt;
1102        (void) snprintf(get_line(0, 0), get_line_remain(),
1103            "Next Header = %u (%s)", nxt_hdr, getproto(nxt_hdr));

1105        while (len > 0) {
1106                data = ndata;
1107                GETINT8(op_type, data);
1108                if (op_type == IP6OPT_PAD1) {
1109                        (void) snprintf(get_line(0, 0), get_line_remain(),
1110                            "pad1 option ");
1111                        len--;
1112                        ndata = data;
1113                        continue;
1114                }
1115                GETINT8(op_len, data);
1116                if (len < 2 || op_len + 2 > len) {
1117                        (void) snprintf(get_line(0, 0), get_line_remain(),
```



```
1118                            "Error: option %u truncated (%u + 2 > %u)",
1119                            op_type, op_len, len);
1120                        op_len = len - 2;
1121                        /*
1122                         * Continue processing the malformed option so that we
1123                         * can display as much as possible.
1124                         */
1125                }

1127                /* advance pointers to the next option */
1128                len -= op_len + 2;
1129                ndata = data + op_len;

1131                /* process this option */
1132                switch (op_type) {
1133                case IP6OPT_PADN:
1134                        (void) snprintf(get_line(0, 0), get_line_remain(),
1135                            "padN option len = %u", op_len);
1136                        break;
1137                case IP6OPT_TUNNEL_LIMIT:
1138                        GETINT8(value, data);
1139                        (void) snprintf(get_line(0, 0), get_line_remain(),
1140                            "tunnel encapsulation limit len = %d, value = %d",
1141                            op_len, value);
1142                        break;
1143                case IP6OPT_LS:
1144                        print_ip6opt_ls(data, op_len);
1145                        break;
1146                default:
1147                        (void) snprintf(get_line(0, 0), get_line_remain(),
1148                            "Option type = %u, len = %u", op_type, op_len);
1149                        break;
1150                }
1151        }

1153        show_space();
1154 }

1156 #define ALABEL_MAXLEN   256

1158 static char ascii_label[ALABEL_MAXLEN];
1159 static char *plabel = ascii_label;

1161 struct snoop_pair {
1162        int val;
1163        const char *name;
1164 };

1166 static struct snoop_pair ripso_class_tbl[] = {
1167        TSOL_CL_TOP_SECRET,     "TOP SECRET",
1168        TSOL_CL_SECRET,         "SECRET",
1169        TSOL_CL_CONFIDENTIAL,   "CONFIDENTIAL",
1170        TSOL_CL_UNCLASSIFIED,   "UNCLASSIFIED",
1171        -1,                     NULL
1172 };

1174 static struct snoop_pair ripso_prot_tbl[] = {
1175        TSOL_PA_GENSER,         "GENSER",
1176        TSOL_PA_SIOP_ESI,       "SIOP-ESI",
1177        TSOL_PA_SCI,            "SCI",
1178        TSOL_PA_NSA,            "NSA",
1179        TSOL_PA_DOE,            "DOE",
1180        0x04,                   "UNASSIGNED",
1181        0x02,                   "UNASSIGNED",
1182        -1,                     NULL
1183 };
```

```
1185 static struct snoop_pair *
1186 get_pair_byval(struct snoop_pair pairlist[], int val)
1187 {
1188         int i;

1190         for (i = 0; pairlist[i].name != NULL; i++)
1191                 if (pairlist[i].val == val)
1192                         return (&pairlist[i]);
1193         return (NULL);
1194 }

1196 static void
1197 print_ripso(const uchar_t *opt)
1198 {
1199         struct snoop_pair *ripso_class;
1200         int i, index, prot_len;
1201         boolean_t first_prot;
1202         char line[100], *ptr;

1204         prot_len = opt[1] - 3;
1205         if (prot_len < 0)
1206                 return;

1208         show_header("RIPSO:  ", "Revised IP Security Option", 0);
1209         show_space();

1211         (void) snprintf(get_line(0, 0), get_line_remain(),
1212             "Type = Basic Security Option (%d), Length = %d", opt[0], opt[1]);

1214         /*
1215          * Display Classification Level
1216          */
1217         ripso_class = get_pair_byval(ripso_class_tbl, (int)opt[2]);
1218         if (ripso_class != NULL)
1219                 (void) snprintf(get_line(0, 0), get_line_remain(),
1220                     "Classification = Unknown (0x%02x)", opt[2]);
1221         else
1222                 (void) snprintf(get_line(0, 0), get_line_remain(),
1223                     "Classification = %s (0x%02x)",
1224                     ripso_class->name, ripso_class->val);

1226         /*
1227          * Display Protection Authority Flags
1228          */
1229         (void) snprintf(line, sizeof (line), "Protection Authority = ");
1230         ptr = line;
1231         first_prot = B_TRUE;
1232         for (i = 0; i < prot_len; i++) {
1233                 index = 0;
1234                 while (ripso_prot_tbl[index].name != NULL) {
1235                         if (opt[3 + i] & ripso_prot_tbl[index].val) {
1236                                 ptr = strchr(ptr, 0);
1237                                 if (!first_prot) {
1238                                         (void) strlcpy(ptr, ", ",
1239                                             sizeof (line) - (ptr - line));
1240                                         ptr = strchr(ptr, 0);
1241                                 }
1242                                 (void) snprintf(ptr,
1243                                     sizeof (line) - (ptr - line),
1244                                     "%s (0x%02x)",
1245                                     ripso_prot_tbl[index].name,
1246                                     ripso_prot_tbl[index].val);
1247                         }
1248                         index++;
1249                 }
```



```
1250                 if ((opt[3 + i] & 1) == 0)
1251                         break;
1252         }
1253         if (!first_prot)
1254                 (void) snprintf(get_line(0, 0), get_line_remain(), "%s", line);
1255         else
1256                 (void) snprintf(get_line(0, 0), get_line_remain(), "%sNone",
1257                     line);
1258 }

1260 #define CIPSO_GENERIC_ARRAY_LEN 200

1262 /*
1263  * Return 1 if CIPSO SL and Categories are all 1's; 0 otherwise.
1264  *
1265  * Note: opt starts with "Tag Type":
1266  *
1267  * |tag_type(1)|tag_length(1)|align(1)|sl(1)|categories(variable)|
1268  *
1269  */
1270 static boolean_t
1271 cipso_high(const uchar_t *opt)
1272 {
1273         int i;

1275         if (((int)opt[1] + 6) < IP_MAX_OPT_LENGTH)
1276                 return (B_FALSE);
1277         for (i = 0; i < ((int)opt[1] - 3); i++)
1278                 if (opt[3 + i] != 0xff)
1279                         return (B_FALSE);
1280         return (B_TRUE);
1281 }

1283 /*
1284  * Converts CIPSO label to SL.
1285  *
1286  * Note: opt starts with "Tag Type":
1287  *
1288  * |tag_type(1)|tag_length(1)|align(1)|sl(1)|categories(variable)|
1289  *
1290  */
1291 static void
1292 cipso2sl(const uchar_t *opt, bslabel_t *sl, int *high)
1293 {
1294         int i, taglen;
1295         uchar_t *q = (uchar_t *)&((_bslabel_impl_t *)sl)->compartments;

1297         *high = 0;
1298         taglen = opt[1];
1299         memset((caddr_t)sl, 0, sizeof (bslabel_t));

1301         if (cipso_high(opt)) {
1302                 BSLHIGH(sl);
1303                 *high = 1;
1304         } else {
1305                 LCLASS_SET((_bslabel_impl_t *)sl, opt[3]);
1306                 for (i = 0; i < taglen - TSOL_TT1_MIN_LENGTH; i++)
1307                         q[i] = opt[TSOL_TT1_MIN_LENGTH + i];
1308         }
1309         SETBLTYPE(sl, SUN_SL_ID);
1310 }

1312 static int
1313 interpret_cipso_tagtype1(const uchar_t *opt)
1314 {
1315         int i, taglen, ishigh;
```

```
1316         bslabel_t sl;
1317         char line[CIPSO_GENERIC_ARRAY_LEN], *ptr;

1319         taglen = opt[1];
1320         if (taglen < TSOL_TT1_MIN_LENGTH ||
1321             taglen > TSOL_TT1_MAX_LENGTH)
1322                 return (taglen);

1324         (void) snprintf(get_line(0, 0), get_line_remain(),
1325             "Tag Type = %d, Tag Length = %d", opt[0], opt[1]);
1326         (void) snprintf(get_line(0, 0), get_line_remain(),
1327             "Sensitivity Level = 0x%02x", opt[3]);
1328         ptr = line;
1329         for (i = 0; i < taglen - TSOL_TT1_MIN_LENGTH; i++) {
1330                 (void) snprintf(ptr, sizeof (line) - (ptr - line), "%02x",
1331                     opt[TSOL_TT1_MIN_LENGTH + i]);
1332                 ptr = strchr(ptr, 0);
1333         }
1334         if (i != 0) {
1335                 (void) snprintf(get_line(0, 0), get_line_remain(),
1336                     "Categories = ");
1337                 (void) snprintf(get_line(0, 0), get_line_remain(), "\t%s",
1338                     line);
1339         } else {
1340                 (void) snprintf(get_line(0, 0), get_line_remain(),
1341                     "Categories = None");
1342         }
1343         cipso2sl(opt, &sl, &ishigh);
1344         if (is_system_labeled()) {
1345                 if (bsltos(&sl, &plabel, ALABEL_MAXLEN,
1346                     LONG_CLASSIFICATION|LONG_WORDS|VIEW_INTERNAL) < 0) {
1347                         (void) snprintf(get_line(0, 0), get_line_remain(),
1348                             "The Sensitivity Level and Categories can't be "
1349                             "mapped to a valid SL");
1350                 } else {
1351                         (void) snprintf(get_line(0, 0), get_line_remain(),
1352                             "The Sensitivity Level and Categories are mapped "
1353                             "to the SL:");
1354                         (void) snprintf(get_line(0, 0), get_line_remain(),
1355                             "\t%s", ascii_label);
1356                 }
1357         }
1358         return (taglen);
1359 }

1361 /*
1362  * The following struct definition #define's are copied from TS1.x. They are
1363  * not used here (except TTYPE_3_MAX_TOKENS), but included as a reference for
1364  * the tag type 3 packet format.
1365  */
1366 #define TTYPE_3_MAX_TOKENS      7

1368 /*
1369  * Display CIPSO tag type 3 which is defined by MAXSIX.
1370  */
1371 static int
1372 interpret_cipso_tagtype3(const uchar_t *opt)
1373 {
1374         uchar_t tagtype;
1375         int index, numtokens, taglen;
1376         uint16_t mask;
1377         uint32_t token;
1378         static const char *name[] = {
1379                 "SL",
1380                 "NCAV",
1381                 "INTEG",
```



```
1382                 "SID",
1383                 "undefined",
1384                 "undefined",
1385                 "IL",
1386                 "PRIVS",
1387                 "LUID",
1388                 "PID",
1389                 "IDS",
1390                 "ACL"
1391         };

1393         tagtype = *opt++;
1394         (void) memcpy(&mask, opt + 3, sizeof (mask));
1395         (void) snprintf(get_line(0, 0), get_line_remain(),
1396             "Tag Type = %d (MAXSIX)", tagtype);
1397         (void) snprintf(get_line(0, 0), get_line_remain(),
1398             "Generation = 0x%02x%02x%02x, Mask = 0x%04x", opt[0], opt[1],
1399             opt[2], mask);
1400         opt += 3 + sizeof (mask);

1402         /*
1403          * Display tokens
1404          */
1405         numtokens = 0;
1406         index = 0;
1407         while (mask != 0 && numtokens < TTYPE_3_MAX_TOKENS) {
1408                 if (mask & 0x0001) {
1409                         (void) memcpy(&token, opt, sizeof (token));
1410                         opt += sizeof (token);
1411                         (void) snprintf(get_line(0, 0), get_line_remain(),
1412                             "Attribute = %s, Token = 0x%08x",
1413                             index < sizeof (name) / sizeof (*name) ?
1414                             name[index] : "unknown", token);
1415                         numtokens++;
1416                 }
1417                 mask = mask >> 1;
1418                 index++;
1419         }

1421         taglen = 6 + numtokens * 4;
1422         return (taglen);
1423 }

1425 static void
1426 print_cipso(const uchar_t *opt)
1427 {
1428         int optlen, taglen, tagnum;
1429         uint32_t doi;
1430         char line[CIPSO_GENERIC_ARRAY_LEN];
1431         char *oldnest;

1433         optlen = opt[1];
1434         if (optlen < TSOL_CIPSO_MIN_LENGTH || optlen > TSOL_CIPSO_MAX_LENGTH)
1435                 return;

1437         oldnest = prot_nest_prefix;
1438         prot_nest_prefix = prot_prefix;
1439         show_header("CIPSO:  ", "Common IP Security Option", 0);
1440         show_space();

1442         /*
1443          * Display CIPSO Header
1444          */
1445         (void) snprintf(get_line(0, 0), get_line_remain(),
1446             "Type = CIPSO (%d), Length = %d", opt[0], opt[1]);
1447         (void) memcpy(&doi, opt + 2, sizeof (doi));
```

```
1448         (void) snprintf(get_line(0, 0), get_line_remain(),
1449             "Domain of Interpretation = %u", (unsigned)ntohl(doi));

1451         if (opt[1] == TSOL_CIPSO_MIN_LENGTH) {  /* no tags */
1452                 show_space();
1453                 prot_prefix = prot_nest_prefix;
1454                 prot_nest_prefix = oldnest;
1455                 return;
1456         }
1457         optlen -= TSOL_CIPSO_MIN_LENGTH;
1458         opt += TSOL_CIPSO_MIN_LENGTH;

1460         /*
1461          * Display Each Tag
1462          */
1463         tagnum = 1;
1464         while (optlen >= TSOL_TT1_MIN_LENGTH) {
1465                 (void) snprintf(line, sizeof (line), "Tag# %d", tagnum);
1466                 show_header("CIPSO:  ", line, 0);
1467                 /*
1468                  * We handle tag type 1 and 3 only. Note, tag type 3
1469                  * is MAXSIX defined.
1470                  */
1471                 switch (opt[0]) {
1472                 case 1:
1473                         taglen = interpret_cipso_tagtype1(opt);
1474                         break;
1475                 case 3:
1476                         taglen = interpret_cipso_tagtype3(opt);
1477                         break;
1478                 default:
1479                         (void) snprintf(get_line(0, 0), get_line_remain(),
1480                             "Unknown Tag Type %d", opt[0]);
1481                         show_space();
1482                         prot_prefix = prot_nest_prefix;
1483                         prot_nest_prefix = oldnest;
1484                         return;
1485                 }

1487                 /*
1488                  * Move to the next tag
1489                  */
1490                 if (taglen <= 0)
1491                         break;
1492                 optlen -= taglen;
1493                 opt += taglen;
1494                 tagnum++;
1495         }
1496         show_space();
1497         prot_prefix = prot_nest_prefix;
1498         prot_nest_prefix = oldnest;
1499 }
```

```
**********************************************************
    6896 Mon Jul  9 14:38:08 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ipsec.c
dccp: snoop, build system fixes
**********************************************************
______unchanged_portion_omitted_

 102 int
 103 interpret_ah(int flags, uint8_t *hdr, int iplen, int fraglen)
 104 {
 105         /* LINTED: alignment */
 106         ah_t *ah = (ah_t *)hdr;
 107         ah_t *aligned_ah;
 108         ah_t storage;   /* In case hdr isn't aligned. */
 109         char *line, *buff;
 110         uint_t ahlen, auth_data_len;
 111         uint8_t *auth_data, *data;
 112         int new_iplen;
 113         uint8_t proto;

 115         if (fraglen < sizeof (ah_t))
 116                 return (fraglen);               /* incomplete header */

 118         if (!IS_P2ALIGNED(hdr, 4)) {
 119                 aligned_ah = (ah_t *)&storage;
 120                 bcopy(hdr, &storage, sizeof (ah_t));
 121         } else {
 122                 aligned_ah = ah;
 123         }

 125         /*
 126          * "+ 8" is for the "constant" part that's not included in the AH
 127          * length.
 128          *
 129          * The AH RFC specifies the length field in "length in 4-byte units,
 130          * not counting the first 8 bytes".  So if an AH is 24 bytes long,
 131          * the length field will contain "4".  (4 * 4 + 8 == 24).
 132          */
 133         ahlen = (aligned_ah->ah_length << 2) + 8;
 134         fraglen -= ahlen;
 135         if (fraglen < 0)
 136                 return (fraglen + ahlen);       /* incomplete header */

 138         auth_data_len = ahlen - sizeof (ah_t);
 139         auth_data = (uint8_t *)(ah + 1);
 140         data = auth_data + auth_data_len;

 142         if (flags & F_SUM) {
 143                 line = (char *)get_sum_line();
 144                 (void) sprintf(line, "AH SPI=0x%x Replay=%u",
 145                     ntohl(aligned_ah->ah_spi), ntohl(aligned_ah->ah_replay));
 146                 line += strlen(line);
 147         }

 149         if (flags & F_DTAIL) {
 150                 show_header("AH:  ", "Authentication Header", ahlen);
 151                 show_space();
 152                 (void) sprintf(get_line((char *)&ah->ah_nexthdr - dlc_header,
 153                     1), "Next header = %d (%s)", aligned_ah->ah_nexthdr,
 154                     getproto(aligned_ah->ah_nexthdr));
 155                 (void) sprintf(get_line((char *)&ah->ah_length - dlc_header, 1),
 156                     "AH length = %d (%d bytes)", aligned_ah->ah_length, ahlen);
 157                 (void) sprintf(get_line((char *)&ah->ah_reserved - dlc_header,
 158                     2), "<Reserved field = 0x%x>",
 159                     ntohs(aligned_ah->ah_reserved));
 160                 (void) sprintf(get_line((char *)&ah->ah_spi - dlc_header, 4),
```



```
 161                     "SPI = 0x%x", ntohl(aligned_ah->ah_spi));
 162                 (void) sprintf(get_line((char *)&ah->ah_replay - dlc_header, 4),
 163                     "Replay = %u", ntohl(aligned_ah->ah_replay));

 165                 /*
 166                  * 2 for two hex digits per auth_data byte
 167                  * plus one byte for trailing null byte.
 168                  */
 169                 buff = malloc(auth_data_len * 2 + 1);
 170                 if (buff != NULL) {
 171                         int i;

 173                         for (i = 0; i < auth_data_len; i++)
 174                                 sprintf(buff + i * 2, "%02x", auth_data[i]);
 175                 }

 177                 (void) sprintf(get_line((char *)auth_data - dlc_header,
 178                     auth_data_len), "ICV = %s",
 179                     (buff == NULL) ? "<out of memory>" : buff);

 181                 /* malloc(3c) says I can call free even if buff == NULL */
 182                 free(buff);

 184                 show_space();
 185         }

 187         new_iplen = iplen - ahlen;
 188         proto = aligned_ah->ah_nexthdr;

 190         /*
 191          * Print IPv6 Extension Headers, or skip them in the summary case.
 192          */
 193         if (proto == IPPROTO_HOPOPTS || proto == IPPROTO_DSTOPTS ||
 194             proto == IPPROTO_ROUTING || proto == IPPROTO_FRAGMENT) {
 195                 (void) print_ipv6_extensions(flags, &data, &proto, &iplen,
 196                     &fraglen);
 197         }

 199         if (fraglen > 0)
 200                 switch (proto) {
 201                         case IPPROTO_ENCAP:
 202                                 /* LINTED: alignment */
 203                                 (void) interpret_ip(flags, (struct ip *)data,
 204                                     new_iplen);
 205                                 break;
 206                         case IPPROTO_IPV6:
 207                                 (void) interpret_ipv6(flags, (ip6_t *)data,
 208                                     new_iplen);
 209                                 break;
 210                         case IPPROTO_ICMP:
 211                                 (void) interpret_icmp(flags,
 212                                     /* LINTED: alignment */
 213                                     (struct icmp *)data, new_iplen, fraglen);
 214                                 break;
 215                         case IPPROTO_ICMPV6:
 216                                 /* LINTED: alignment */
 217                                 (void) interpret_icmpv6(flags, (icmp6_t *)data,
 218                                     new_iplen, fraglen);
 219                                 break;
 220                         case IPPROTO_TCP:
 221                                 (void) interpret_tcp(flags,
 222                                     (struct tcphdr *)data, new_iplen, fraglen);
 223                                 break;

 225                         case IPPROTO_ESP:
 226                                 (void) interpret_esp(flags, data, new_iplen,
```

```
227                                     fraglen);
228                             break;

230                     case IPPROTO_AH:
231                             (void) interpret_ah(flags, data, new_iplen,
232                                 fraglen);
233                             break;

235                     case IPPROTO_UDP:
236                             (void) interpret_udp(flags,
237                                 (struct udphdr *)data, new_iplen, fraglen);
238                             break;
239                     case IPPROTO_DCCP:
240                             (void) interpret_dccp(flags,
241                                 (struct dccphdr *)data, new_iplen, fraglen);
242                             break;
243 #endif /* ! codereview */
244                     /* default case is to not print anything else */
245                 }

247         return (ahlen);
248 }
```

```
**********************************************************
   39796 Mon Jul  9 14:38:08 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_pf.c
dccp: options and features
**********************************************************
______unchanged_portion_omitted_

 140 static transport_table_t ether_transport_mapping_table[] = {
 141         {IPPROTO_TCP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 142         {IPPROTO_TCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 143         {IPPROTO_UDP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 144         {IPPROTO_UDP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 145         {IPPROTO_OSPF, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 146         {IPPROTO_OSPF, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 147         {IPPROTO_SCTP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 148         {IPPROTO_SCTP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 149         {IPPROTO_ICMP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 150         {IPPROTO_ICMPV6, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 151         {IPPROTO_ENCAP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 152         {IPPROTO_ESP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 153         {IPPROTO_ESP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 154         {IPPROTO_AH, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 155         {IPPROTO_AH, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 156         {IPPROTO_DCCP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 157         {IPPROTO_DCCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 158 #endif /* ! codereview */
 159         {-1, 0, 0}      /* must be the final entry */
 160 };

 162 static transport_table_t ipnet_transport_mapping_table[] = {
 163         {IPPROTO_TCP, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 164             IPV4_TYPE_HEADER_OFFSET},
 165         {IPPROTO_TCP, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 166             IPV6_TYPE_HEADER_OFFSET},
 167         {IPPROTO_UDP, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 168             IPV4_TYPE_HEADER_OFFSET},
 169         {IPPROTO_UDP, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 170             IPV6_TYPE_HEADER_OFFSET},
 171         {IPPROTO_OSPF, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 172             IPV4_TYPE_HEADER_OFFSET},
 173         {IPPROTO_OSPF, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 174             IPV6_TYPE_HEADER_OFFSET},
 175         {IPPROTO_SCTP, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 176             IPV4_TYPE_HEADER_OFFSET},
 177         {IPPROTO_SCTP, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 178             IPV6_TYPE_HEADER_OFFSET},
 179         {IPPROTO_ICMP, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 180             IPV4_TYPE_HEADER_OFFSET},
 181         {IPPROTO_ICMPV6, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 182             IPV6_TYPE_HEADER_OFFSET},
 183         {IPPROTO_ENCAP, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 184             IPV4_TYPE_HEADER_OFFSET},
 185         {IPPROTO_ESP, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 186             IPV4_TYPE_HEADER_OFFSET},
 187         {IPPROTO_ESP, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 188             IPV6_TYPE_HEADER_OFFSET},
 189         {IPPROTO_AH, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 190             IPV4_TYPE_HEADER_OFFSET},
 191         {IPPROTO_AH, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 192             IPV6_TYPE_HEADER_OFFSET},
 193         {IPPROTO_DCCP, (DL_IPNETINFO_VERSION << 8 | AF_INET),
 194             IPV4_TYPE_HEADER_OFFSET},
 195         {IPPROTO_DCCP, (DL_IPNETINFO_VERSION << 8 | AF_INET6),
 196             IPV6_TYPE_HEADER_OFFSET},
 197 #endif /* ! codereview */
 198         {-1, 0, 0}      /* must be the final entry */
```



```
 199 };

 201 static transport_table_t ib_transport_mapping_table[] = {
 202         {IPPROTO_TCP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 203         {IPPROTO_TCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 204         {IPPROTO_UDP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 205         {IPPROTO_UDP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 206         {IPPROTO_OSPF, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 207         {IPPROTO_OSPF, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 208         {IPPROTO_SCTP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 209         {IPPROTO_SCTP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 210         {IPPROTO_ICMP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 211         {IPPROTO_ICMPV6, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 212         {IPPROTO_ENCAP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 213         {IPPROTO_ESP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 214         {IPPROTO_ESP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 215         {IPPROTO_AH, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 216         {IPPROTO_AH, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 217         {IPPROTO_DCCP, ETHERTYPE_IP,   IPV4_TYPE_HEADER_OFFSET},
 218         {IPPROTO_DCCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
 219 #endif /* ! codereview */
 220         {-1, 0, 0}      /* must be the final entry */
 221 };

 223 typedef struct datalink {
 224         uint_t                  dl_type;
 225         void                    (*dl_match_fn)(uint_t datatype);
 226         transport_table_t       *dl_trans_map_tbl;
 227         network_table_t         *dl_net_map_tbl;
 228         int                     dl_link_header_len;
 229         int                     dl_link_type_offset;
 230         int                     dl_link_dest_offset;
 231         int                     dl_link_src_offset;
 232         int                     dl_link_addr_len;
 233 } datalink_t;

 235 datalink_t      dl;

 237 #define IPV4_SRCADDR_OFFSET     (dl.dl_link_header_len + 12)
 238 #define IPV4_DSTADDR_OFFSET     (dl.dl_link_header_len + 16)
 239 #define IPV6_SRCADDR_OFFSET     (dl.dl_link_header_len + 8)
 240 #define IPV6_DSTADDR_OFFSET     (dl.dl_link_header_len + 24)

 242 #define IPNET_SRCZONE_OFFSET 16
 243 #define IPNET_DSTZONE_OFFSET 20

 245 static int inBrace = 0, inBraceOR = 0;
 246 static int foundOR = 0;
 247 char *tkp, *sav_tkp;
 248 char *token;
 249 enum { EOL, ALPHA, NUMBER, FIELD, ADDR_IP, ADDR_ETHER, SPECIAL,
 250         ADDR_IP6 } tokentype;
 251 uint_t tokenval;

 253 enum direction { ANY, TO, FROM };
 254 enum direction dir;

 256 extern void next();

 258 static void pf_expression();
 259 static void pf_check_vlan_tag(uint_t offset);
 260 static void pf_clear_offset_register();
 261 static void pf_emit_load_offset(uint_t offset);
 262 static void pf_match_ethertype(uint_t ethertype);
 263 static void pf_match_ipnettype(uint_t type);
 264 static void pf_match_ibtype(uint_t type);
```

```
265 static void pf_check_transport_protocol(uint_t transport_protocol);
266 static void pf_compare_value_mask_generic(int offset, uint_t len,
267     uint_t val, int mask, uint_t op);
268 static void pf_matchfn(const char *name);

270 /*
271  * This pointer points to the function that last generated
272  * instructions to change the offset register.  It's used
273  * for comparisons to see if we need to issue more instructions
274  * to change the register.
275  *
276  * It's initialized to pf_clear_offset_register because the offset
277  * register in pfmod is initialized to zero, similar to the state
278  * it would be in after executing the instructions issued by
279  * pf_clear_offset_register.
280  */
281 static void *last_offset_operation = (void*)pf_clear_offset_register;

283 static void
284 pf_emit(x)
285         ushort_t x;
286 {
287         if (pfp > &pf.Pf_Filter[PF_MAXFILTERS - 1])
288                 longjmp(env, 1);
289         *pfp++ = x;
290 }

292 static void
293 pf_codeprint(code, len)
294         ushort_t *code;
295         int len;
296 {
297         ushort_t *pc;
298         ushort_t *plast = code + len;
299         int op, action;

301         if (len > 0) {
302                 printf("Kernel Filter:\n");
303         }

305         for (pc = code; pc < plast; pc++) {
306                 printf("\t%3d: ", pc - code);

308                 op = *pc & 0xfc00;      /* high 10 bits */
309                 action = *pc & 0x3ff;   /* low   6 bits */

311                 switch (action) {
312                 case ENF_PUSHLIT:
313                         printf("PUSHLIT ");
314                         break;
315                 case ENF_PUSHZERO:
316                         printf("PUSHZERO ");
317                         break;
318 #ifdef ENF_PUSHONE
319                 case ENF_PUSHONE:
320                         printf("PUSHONE ");
321                         break;
322 #endif
323 #ifdef ENF_PUSHFFFF
324                 case ENF_PUSHFFFF:
325                         printf("PUSHFFFF ");
326                         break;
327 #endif
328 #ifdef ENF_PUSHFF00
329                 case ENF_PUSHFF00:
330                         printf("PUSHFF00 ");
```



```
331                         break;
332 #endif
333 #ifdef ENF_PUSH00FF
334                 case ENF_PUSH00FF:
335                         printf("PUSH00FF ");
336                         break;
337 #endif
338                 case ENF_LOAD_OFFSET:
339                         printf("LOAD_OFFSET ");
340                         break;
341                 case ENF_BRTR:
342                         printf("BRTR ");
343                         break;
344                 case ENF_BRFL:
345                         printf("BRFL ");
346                         break;
347                 case ENF_POP:
348                         printf("POP ");
349                         break;
350                 }

352                 if (action >= ENF_PUSHWORD)
353                         printf("PUSHWORD %d ", action - ENF_PUSHWORD);

355                 switch (op) {
356                 case ENF_EQ:
357                         printf("EQ ");
358                         break;
359                 case ENF_LT:
360                         printf("LT ");
361                         break;
362                 case ENF_LE:
363                         printf("LE ");
364                         break;
365                 case ENF_GT:
366                         printf("GT ");
367                         break;
368                 case ENF_GE:
369                         printf("GE ");
370                         break;
371                 case ENF_AND:
372                         printf("AND ");
373                         break;
374                 case ENF_OR:
375                         printf("OR ");
376                         break;
377                 case ENF_XOR:
378                         printf("XOR ");
379                         break;
380                 case ENF_COR:
381                         printf("COR ");
382                         break;
383                 case ENF_CAND:
384                         printf("CAND ");
385                         break;
386                 case ENF_CNOR:
387                         printf("CNOR ");
388                         break;
389                 case ENF_CNAND:
390                         printf("CNAND ");
391                         break;
392                 case ENF_NEQ:
393                         printf("NEQ ");
394                         break;
395                 }
```

```
397                 if (action == ENF_PUSHLIT ||
398                     action == ENF_LOAD_OFFSET ||
399                     action == ENF_BRTR ||
400                     action == ENF_BRFL) {
401                         pc++;
402                         printf("\n\t%3d:   %d (0x%04x)", pc - code, *pc, *pc);
403                 }

405                 printf("\n");
406         }
407 }

409 /*
410  * Emit packet filter code to check a
411  * field in the packet for a particular value.
412  * Need different code for each field size.
413  * Since the pf can only compare 16 bit quantities
414  * we have to use masking to compare byte values.
415  * Long word (32 bit) quantities have to be done
416  * as two 16 bit comparisons.
417  */
418 static void
419 pf_compare_value(int offset, uint_t len, uint_t val)
420 {
421         /*
422          * If the property being filtered on is absent in the media
423          * packet, error out.
424          */
425         if (offset == -1)
426                 pr_err("filter option unsupported on media");

428         switch (len) {
429         case 1:
430                 pf_emit(ENF_PUSHWORD + offset / 2);
431 #if defined(_BIG_ENDIAN)
432                 if (offset % 2)
433 #else
434                 if (!(offset % 2))
435 #endif
436                 {
437 #ifdef ENF_PUSH00FF
438                         pf_emit(ENF_PUSH00FF | ENF_AND);
439 #else
440                         pf_emit(ENF_PUSHLIT | ENF_AND);
441                         pf_emit(0x00FF);
442 #endif
443                         pf_emit(ENF_PUSHLIT | ENF_EQ);
444                         pf_emit(val);
445                 } else {
446 #ifdef ENF_PUSHFF00
447                         pf_emit(ENF_PUSHFF00 | ENF_AND);
448 #else
449                         pf_emit(ENF_PUSHLIT | ENF_AND);
450                         pf_emit(0xFF00);
451 #endif
452                         pf_emit(ENF_PUSHLIT | ENF_EQ);
453                         pf_emit(val << 8);
454                 }
455                 break;

457         case 2:
458                 pf_emit(ENF_PUSHWORD + offset / 2);
459                 pf_emit(ENF_PUSHLIT | ENF_EQ);
460                 pf_emit((ushort_t)val);
461                 break;
```



```
463         case 4:
464                 pf_emit(ENF_PUSHWORD + offset / 2);
465                 pf_emit(ENF_PUSHLIT | ENF_EQ);
466 #if defined(_BIG_ENDIAN)
467                 pf_emit(val >> 16);
468 #elif defined(_LITTLE_ENDIAN)
469                 pf_emit(val & 0xffff);
470 #else
471 #error One of _BIG_ENDIAN and _LITTLE_ENDIAN must be defined
472 #endif
473                 pf_emit(ENF_PUSHWORD + (offset / 2) + 1);
474                 pf_emit(ENF_PUSHLIT | ENF_EQ);
475 #if defined(_BIG_ENDIAN)
476                 pf_emit(val & 0xffff);
477 #else
478                 pf_emit(val >> 16);
479 #endif
480                 pf_emit(ENF_AND);
481                 break;
482         }
483 }

485 /*
486  * same as pf_compare_value, but only for emiting code to
487  * compare ipv6 addresses.
488  */
489 static void
490 pf_compare_value_v6(int offset, uint_t len, struct in6_addr val)
491 {
492         int i;

494         for (i = 0; i < len; i += 2) {
495                 pf_emit(ENF_PUSHWORD + offset / 2 + i / 2);
496                 pf_emit(ENF_PUSHLIT | ENF_EQ);
497                 pf_emit(*(uint16_t *)&val.s6_addr[i]);
498                 if (i != 0)
499                         pf_emit(ENF_AND);
500         }
501 }


504 /*
505  * Same as above except mask the field value
506  * before doing the comparison.  The comparison checks
507  * to make sure the values are equal.
508  */
509 static void
510 pf_compare_value_mask(int offset, uint_t len, uint_t val, int mask)
511 {
512         pf_compare_value_mask_generic(offset, len, val, mask, ENF_EQ);
513 }

515 /*
516  * Same as above except the values are compared to see if they are not
517  * equal.
518  */
519 static void
520 pf_compare_value_mask_neq(int offset, uint_t len, uint_t val, int mask)
521 {
522         pf_compare_value_mask_generic(offset, len, val, mask, ENF_NEQ);
523 }

525 /*
526  * Similar to pf_compare_value.
527  *
528  * This is the utility function that does the actual work to compare
```

```
529  * two values using a mask.  The comparison operation is passed into
530  * the function.
531  */
532 static void
533 pf_compare_value_mask_generic(int offset, uint_t len, uint_t val, int mask,
534     uint_t op)
535 {
536         /*
537          * If the property being filtered on is absent in the media
538          * packet, error out.
539          */
540         if (offset == -1)
541                 pr_err("filter option unsupported on media");

543         switch (len) {
544         case 1:
545                 pf_emit(ENF_PUSHWORD + offset / 2);
546 #if defined(_BIG_ENDIAN)
547                 if (offset % 2)
548 #else
549                 if (!offset % 2)
550 #endif
551                 {
552                         pf_emit(ENF_PUSHLIT | ENF_AND);
553                         pf_emit(mask & 0x00ff);
554                         pf_emit(ENF_PUSHLIT | op);
555                         pf_emit(val);
556                 } else {
557                         pf_emit(ENF_PUSHLIT | ENF_AND);
558                         pf_emit((mask << 8) & 0xff00);
559                         pf_emit(ENF_PUSHLIT | op);
560                         pf_emit(val << 8);
561                 }
562                 break;

564         case 2:
565                 pf_emit(ENF_PUSHWORD + offset / 2);
566                 pf_emit(ENF_PUSHLIT | ENF_AND);
567                 pf_emit(htons((ushort_t)mask));
568                 pf_emit(ENF_PUSHLIT | op);
569                 pf_emit(htons((ushort_t)val));
570                 break;

572         case 4:
573                 pf_emit(ENF_PUSHWORD + offset / 2);
574                 pf_emit(ENF_PUSHLIT | ENF_AND);
575                 pf_emit(htons((ushort_t)((mask >> 16) & 0xffff)));
576                 pf_emit(ENF_PUSHLIT | op);
577                 pf_emit(htons((ushort_t)((val >> 16) & 0xffff)));

579                 pf_emit(ENF_PUSHWORD + (offset / 2) + 1);
580                 pf_emit(ENF_PUSHLIT | ENF_AND);
581                 pf_emit(htons((ushort_t)(mask & 0xffff)));
582                 pf_emit(ENF_PUSHLIT | op);
583                 pf_emit(htons((ushort_t)(val & 0xffff)));

585                 pf_emit(ENF_AND);
586                 break;
587         }
588 }

590 /*
591  * Like pf_compare_value() but compare on a 32-bit zoneid value.
592  * The argument val passed in is in network byte order.
593  */
594 static void
```



```
595 pf_compare_zoneid(int offset, uint32_t val)
596 {
597         int i;

599         for (i = 0; i < sizeof (uint32_t) / 2; i ++) {
600                 pf_emit(ENF_PUSHWORD + offset / 2 + i);
601                 pf_emit(ENF_PUSHLIT | ENF_EQ);
602                 pf_emit(((uint16_t *)&val)[i]);
603                 if (i != 0)
604                         pf_emit(ENF_AND);
605         }
606 }

608 /*
609  * Generate pf code to match an IPv4 or IPv6 address.
610  */
611 static void
612 pf_ipaddr_match(which, hostname, inet_type)
613         enum direction which;
614         char *hostname;
615         int inet_type;
616 {
617         bool_t found_host;
618         uint_t *addr4ptr;
619         uint_t addr4;
620         struct in6_addr *addr6ptr;
621         int h_addr_index;
622         struct hostent *hp = NULL;
623         int error_num = 0;
624         boolean_t first = B_TRUE;
625         int pass = 0;
626         int i;

628         /*
629          * The addr4offset and addr6offset variables simplify the code which
630          * generates the address comparison filter.  With these two variables,
631          * duplicate code need not exist for the TO and FROM case.
632          * A value of -1 describes the ANY case (TO and FROM).
633          */
634         int addr4offset;
635         int addr6offset;

637         found_host = 0;

639         if (tokentype == ADDR_IP) {
640                 hp = getipnodebyname(hostname, AF_INET, 0, &error_num);
641                 if (hp == NULL) {
642                         if (error_num == TRY_AGAIN) {
643                                 pr_err("could not resolve %s (try again later)",
644                                     hostname);
645                         } else {
646                                 pr_err("could not resolve %s", hostname);
647                         }
648                 }
649                 inet_type = IPV4_ONLY;
650         } else if (tokentype == ADDR_IP6) {
651                 hp = getipnodebyname(hostname, AF_INET6, 0, &error_num);
652                 if (hp == NULL) {
653                         if (error_num == TRY_AGAIN) {
654                                 pr_err("could not resolve %s (try again later)",
655                                     hostname);
656                         } else {
657                                 pr_err("could not resolve %s", hostname);
658                         }
659                 }
660                 inet_type = IPV6_ONLY;
```

```
661         } else if (tokentype == ALPHA) {
662                 /* Some hostname i.e. tokentype is ALPHA */
663                 switch (inet_type) {
664                 case IPV4_ONLY:
665                         /* Only IPv4 address is needed */
666                         hp = getipnodebyname(hostname, AF_INET, 0, &error_num);
667                         if (hp != NULL) {
668                                 found_host = 1;
669                         }
670                         break;
671                 case IPV6_ONLY:
672                         /* Only IPv6 address is needed */
673                         hp = getipnodebyname(hostname, AF_INET6, 0, &error_num);
674                         if (hp != NULL) {
675                                 found_host = 1;
676                         }
677                         break;
678                 case IPV4_AND_IPV6:
679                         /* Both IPv4 and IPv6 are needed */
680                         hp = getipnodebyname(hostname, AF_INET6,
681                             AI_ALL | AI_V4MAPPED, &error_num);
682                         if (hp != NULL) {
683                                 found_host = 1;
684                         }
685                         break;
686                 default:
687                         found_host = 0;
688                 }

690                 if (!found_host) {
691                         if (error_num == TRY_AGAIN) {
692                                 pr_err("could not resolve %s (try again later)",
693                                     hostname);
694                         } else {
695                                 pr_err("could not resolve %s", hostname);
696                         }
697                 }
698         } else {
699                 pr_err("unknown token type: %s", hostname);
700         }

702         switch (which) {
703         case TO:
704                 addr4offset = IPV4_DSTADDR_OFFSET;
705                 addr6offset = IPV6_DSTADDR_OFFSET;
706                 break;
707         case FROM:
708                 addr4offset = IPV4_SRCADDR_OFFSET;
709                 addr6offset = IPV6_SRCADDR_OFFSET;
710                 break;
711         case ANY:
712                 addr4offset = -1;
713                 addr6offset = -1;
714                 break;
715         }

717         if (hp != NULL && hp->h_addrtype == AF_INET) {
718                 pf_matchfn("ip");
719                 if (dl.dl_type == DL_ETHER)
720                         pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);
721                 h_addr_index = 0;
722                 addr4ptr = (uint_t *)hp->h_addr_list[h_addr_index];
723                 while (addr4ptr != NULL) {
724                         if (addr4offset == -1) {
725                                 pf_compare_value(IPV4_SRCADDR_OFFSET, 4,
726                                     *addr4ptr);
```



```
727                                 if (h_addr_index != 0)
728                                         pf_emit(ENF_OR);
729                                 pf_compare_value(IPV4_DSTADDR_OFFSET, 4,
730                                     *addr4ptr);
731                                 pf_emit(ENF_OR);
732                         } else {
733                                 pf_compare_value(addr4offset, 4,
734                                     *addr4ptr);
735                                 if (h_addr_index != 0)
736                                         pf_emit(ENF_OR);
737                         }
738                         addr4ptr = (uint_t *)hp->h_addr_list[++h_addr_index];
739                 }
740                 pf_emit(ENF_AND);
741         } else {
742                 /* first pass: IPv4 addresses */
743                 h_addr_index = 0;
744                 addr6ptr = (struct in6_addr *)hp->h_addr_list[h_addr_index];
745                 first = B_TRUE;
746                 while (addr6ptr != NULL) {
747                         if (IN6_IS_ADDR_V4MAPPED(addr6ptr)) {
748                                 if (first) {
749                                         pf_matchfn("ip");
750                                         if (dl.dl_type == DL_ETHER) {
751                                                 pf_check_vlan_tag(
752                                                     ENCAP_ETHERTYPE_OFF/2);
753                                         }
754                                         pass++;
755                                 }
756                                 IN6_V4MAPPED_TO_INADDR(addr6ptr,
757                                     (struct in_addr *)&addr4);
758                                 if (addr4offset == -1) {
759                                         pf_compare_value(IPV4_SRCADDR_OFFSET, 4,
760                                             addr4);
761                                         if (!first)
762                                                 pf_emit(ENF_OR);
763                                         pf_compare_value(IPV4_DSTADDR_OFFSET, 4,
764                                             addr4);
765                                         pf_emit(ENF_OR);
766                                 } else {
767                                         pf_compare_value(addr4offset, 4,
768                                             addr4);
769                                         if (!first)
770                                                 pf_emit(ENF_OR);
771                                 }
772                                 if (first)
773                                         first = B_FALSE;
774                         }
775                         addr6ptr = (struct in6_addr *)
776                                 hp->h_addr_list[++h_addr_index];
777                 }
778                 if (!first) {
779                         pf_emit(ENF_AND);
780                 }
781                 /* second pass: IPv6 addresses */
782                 h_addr_index = 0;
783                 addr6ptr = (struct in6_addr *)hp->h_addr_list[h_addr_index];
784                 first = B_TRUE;
785                 while (addr6ptr != NULL) {
786                         if (!IN6_IS_ADDR_V4MAPPED(addr6ptr)) {
787                                 if (first) {
788                                         pf_matchfn("ip6");
789                                         if (dl.dl_type == DL_ETHER) {
790                                                 pf_check_vlan_tag(
791                                                         ENCAP_ETHERTYPE_OFF/2);
792                                         }
```

```
793                                     pass++;
794                             }
795                             if (addr6offset == -1) {
796                                     pf_compare_value_v6(IPV6_SRCADDR_OFFSET,
797                                         16, *addr6ptr);
798                                     if (!first)
799                                             pf_emit(ENF_OR);
800                                     pf_compare_value_v6(IPV6_DSTADDR_OFFSET,
801                                         16, *addr6ptr);
802                                     pf_emit(ENF_OR);
803                             } else {
804                                     pf_compare_value_v6(addr6offset, 16,
805                                         *addr6ptr);
806                                     if (!first)
807                                             pf_emit(ENF_OR);
808                             }
809                             if (first)
810                                     first = B_FALSE;
811                     }
812                     addr6ptr = (struct in6_addr *)
813                             hp->h_addr_list[++h_addr_index];
814             }
815             if (!first) {
816                     pf_emit(ENF_AND);
817             }
818             if (pass == 2) {
819                     pf_emit(ENF_OR);
820             }
821     }

823     if (hp != NULL) {
824             freehostent(hp);
825     }
826 }


829 static void
830 pf_compare_address(int offset, uint_t len, uchar_t *addr)
831 {
832     uint32_t val;
833     uint16_t sval;
834     boolean_t didone = B_FALSE;

836     /*
837      * If the property being filtered on is absent in the media
838      * packet, error out.
839      */
840     if (offset == -1)
841             pr_err("filter option unsupported on media");

843     while (len > 0) {
844             if (len >= 4) {
845                     (void) memcpy(&val, addr, 4);
846                     pf_compare_value(offset, 4, val);
847                     addr += 4;
848                     offset += 4;
849                     len -= 4;
850             } else if (len >= 2) {
851                     (void) memcpy(&sval, addr, 2);
852                     pf_compare_value(offset, 2, sval);
853                     addr += 2;
854                     offset += 2;
855                     len -= 2;
856             } else {
857                     pf_compare_value(offset++, 1, *addr++);
858                     len--;
```



```
859             }
860             if (didone)
861                     pf_emit(ENF_AND);
862             didone = B_TRUE;
863     }
864 }

866 /*
867  * Compare ethernet addresses.
868  */
869 static void
870 pf_etheraddr_match(which, hostname)
871     enum direction which;
872     char *hostname;
873 {
874     struct ether_addr e, *ep = NULL;

876     if (isxdigit(*hostname))
877             ep = ether_aton(hostname);
878     if (ep == NULL) {
879             if (ether_hostton(hostname, &e))
880                     if (!arp_for_ether(hostname, &e))
881                             pr_err("cannot obtain ether addr for %s",
882                                 hostname);
883             ep = &e;
884     }

886     pf_clear_offset_register();

888     switch (which) {
889     case TO:
890             pf_compare_address(dl.dl_link_dest_offset, dl.dl_link_addr_len,
891                 (uchar_t *)ep);
892             break;
893     case FROM:
894             pf_compare_address(dl.dl_link_src_offset, dl.dl_link_addr_len,
895                 (uchar_t *)ep);
896             break;
897     case ANY:
898             pf_compare_address(dl.dl_link_dest_offset, dl.dl_link_addr_len,
899                 (uchar_t *)ep);
900             pf_compare_address(dl.dl_link_src_offset, dl.dl_link_addr_len,
901                 (uchar_t *)ep);
902             pf_emit(ENF_OR);
903             break;
904     }
905 }

907 /*
908  * Emit code to compare the network part of
909  * an IP address.
910  */
911 static void
912 pf_netaddr_match(which, netname)
913     enum direction which;
914     char *netname;
915 {
916     uint_t addr;
917     uint_t mask = 0xff000000;
918     struct netent *np;

920     if (isdigit(*netname)) {
921             addr = inet_network(netname);
922     } else {
923             np = getnetbyname(netname);
924             if (np == NULL)
```

```
925                         pr_err("net %s not known", netname);
926                 addr = np->n_net;
927         }

929         /*
930          * Left justify the address and figure
931          * out a mask based on the supplied address.
932          * Set the mask according to the number of zero
933          * low-order bytes.
934          * Note: this works only for whole octet masks.
935          */
936         if (addr) {
937                 while ((addr & ~mask) != 0) {
938                         mask |= (mask >> 8);
939                 }
940         }

942         pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);

944         switch (which) {
945         case TO:
946                 pf_compare_value_mask(IPV4_DSTADDR_OFFSET, 4, addr, mask);
947                 break;
948         case FROM:
949                 pf_compare_value_mask(IPV4_SRCADDR_OFFSET, 4, addr, mask);
950                 break;
951         case ANY:
952                 pf_compare_value_mask(IPV4_SRCADDR_OFFSET, 4, addr, mask);
953                 pf_compare_value_mask(IPV4_DSTADDR_OFFSET, 4, addr, mask);
954                 pf_emit(ENF_OR);
955                 break;
956         }
957 }

959 /*
960  * Emit code to match on src or destination zoneid.
961  * The zoneid passed in is in network byte order.
962  */
963 static void
964 pf_match_zone(enum direction which, uint32_t zoneid)
965 {
966         if (dl.dl_type != DL_IPNET)
967                 pr_err("zone filter option unsupported on media");

969         switch (which) {
970         case TO:
971                 pf_compare_zoneid(IPNET_DSTZONE_OFFSET, zoneid);
972                 break;
973         case FROM:
974                 pf_compare_zoneid(IPNET_SRCZONE_OFFSET, zoneid);
975                 break;
976         case ANY:
977                 pf_compare_zoneid(IPNET_SRCZONE_OFFSET, zoneid);
978                 pf_compare_zoneid(IPNET_DSTZONE_OFFSET, zoneid);
979                 pf_emit(ENF_OR);
980                 break;
981         }
982 }

984 /*
985  * A helper function to keep the code to emit instructions
986  * to change the offset register in one place.
987  *
988  * INPUTS: offset - An value representing an offset in 16-bit
989  *                  words.
990  * OUTPUTS:  If there is enough room in the storage for the
```



```
991  *           packet filtering program, instructions to load
992  *           a constant to the offset register.  Otherwise,
993  *           nothing.
994  */
995 static void
996 pf_emit_load_offset(uint_t offset)
997 {
998         pf_emit(ENF_LOAD_OFFSET | ENF_NOP);
999         pf_emit(offset);
1000 }

1002 /*
1003  * Clear pfmod's offset register.
1004  *
1005  * INPUTS:  none
1006  * OUTPUTS:  Instructions to clear the offset register if
1007  *           there is enough space remaining in the packet
1008  *           filtering program structure's storage, and
1009  *           the last thing done to the offset register was
1010  *           not clearing the offset register.  Otherwise,
1011  *           nothing.
1012  */
1013 static void
1014 pf_clear_offset_register()
1015 {
1016         if (last_offset_operation != (void*)pf_clear_offset_register) {
1017                 pf_emit_load_offset(0);
1018                 last_offset_operation = (void*)pf_clear_offset_register;
1019         }
1020 }

1022 /*
1023  * This function will issue opcodes to check if a packet
1024  * is VLAN tagged, and if so, update the offset register
1025  * with the appropriate offset.
1026  *
1027  * Note that if the packet is not VLAN tagged, then the offset
1028  * register will be cleared.
1029  *
1030  * If the interface type is not an ethernet type, then this
1031  * function returns without doing anything.
1032  *
1033  * If the last attempt to change the offset register occured because
1034  * of a call to this function that was called with the same offset,
1035  * then we don't issue packet filtering instructions.
1036  *
1037  * INPUTS:  offset - an offset in 16 bit words.  The function
1038  *                   will set the offset register to this
1039  *                   value if the packet is VLAN tagged.
1040  * OUTPUTS:  If the conditions are met, packet filtering instructions.
1041  */
1042 static void
1043 pf_check_vlan_tag(uint_t offset)
1044 {
1045         static uint_t last_offset = 0;

1047         if ((interface->mac_type == DL_ETHER ||
1048             interface->mac_type == DL_CSMACD) &&
1049             (last_offset_operation != (void*)pf_check_vlan_tag ||
1050             last_offset != offset)) {
1051                 /*
1052                  * First thing is to clear the offset register.
1053                  * We don't know what state it is in, and if it
1054                  * is not zero, then we have no idea what we load
1055                  * when we execute ENF_PUSHWORD.
1056                  */
```

```
1057                pf_clear_offset_register();
1058
1059                /*
1060                 * Check the ethertype.
1061                 */
1062                pf_compare_value(dl.dl_link_type_offset, 2,
1063                    htons(ETHERTYPE_VLAN));
1064
1065                /*
1066                 * And if it's not VLAN, don't load offset to the offset
1067                 * register.
1068                 */
1069                pf_emit(ENF_BRFL | ENF_NOP);
1070                pf_emit(3);
1071
1072                /*
1073                 * Otherwise, load offset to the offset register.
1074                 */
1075                pf_emit_load_offset(offset);
1076
1077                /*
1078                 * Now get rid of the results of the comparison,
1079                 * we don't want the results of the comparison to affect
1080                 * other logic in the packet filtering program.
1081                 */
1082                pf_emit(ENF_POP | ENF_NOP);
1083
1084                /*
1085                 * Set the last operation at the end, or any time
1086                 * after the call to pf_clear_offset because
1087                 * pf_clear_offset uses it.
1088                 */
1089                last_offset_operation = (void*)pf_check_vlan_tag;
1090                last_offset = offset;
1091        }
1092 }
1093
1094 /*
1095  * Utility function used to emit packet filtering code
1096  * to match an ethertype.
1097  *
1098  * INPUTS:  ethertype - The ethertype we want to check for.
1099  *                      Don't call htons on the ethertype before
1100  *                      calling this function.
1101  * OUTPUTS:  If there is sufficient storage available, packet
1102  *           filtering code to check an ethertype.  Otherwise,
1103  *           nothing.
1104  */
1105 static void
1106 pf_match_ethertype(uint_t ethertype)
1107 {
1108         /*
1109          * If the user wants to filter on ethertype VLAN,
1110          * then clear the offset register so that the offset
1111          * for ENF_PUSHWORD points to the right place in the
1112          * packet.
1113          *
1114          * Otherwise, call pf_check_vlan_tag to set the offset
1115          * register such that the contents of the offset register
1116          * plus the argument for ENF_PUSHWORD point to the right
1117          * part of the packet, whether or not the packet is VLAN
1118          * tagged.  We call pf_check_vlan_tag with an offset of
1119          * two words because if the packet is VLAN tagged, we have
1120          * to move past the ethertype in the ethernet header, and
1121          * past the lower two octets of the VLAN header to get to
1122          * the ethertype in the VLAN header.
```



```
1123          */
1124         if (ethertype == ETHERTYPE_VLAN)
1125                 pf_clear_offset_register();
1126         else
1127                 pf_check_vlan_tag(2);
1128
1129         pf_compare_value(dl.dl_link_type_offset, 2, htons(ethertype));
1130 }
1131
1132 static void
1133 pf_match_ipnettype(uint_t type)
1134 {
1135         pf_compare_value(dl.dl_link_type_offset, 2, htons(type));
1136 }
1137
1138 static void
1139 pf_match_ibtype(uint_t type)
1140 {
1141         pf_compare_value(dl.dl_link_type_offset, 2, htons(type));
1142 }
1143
1144 /*
1145  * This function uses the table above to generate a
1146  * piece of a packet filtering program to check a transport
1147  * protocol type.
1148  *
1149  * INPUTS:  tranport_protocol - the transport protocol we're
1150  *                              interested in.
1151  * OUTPUTS:  If there is sufficient storage, then packet filtering
1152  *           code to check a transport protocol type.  Otherwise,
1153  *           nothing.
1154  */
1155 static void
1156 pf_check_transport_protocol(uint_t transport_protocol)
1157 {
1158         int i;
1159         uint_t number_of_matches = 0;
1160
1161         for (i = 0; dl.dl_trans_map_tbl[i].transport_protocol != -1; i++) {
1162                 if (transport_protocol ==
1163                     (uint_t)dl.dl_trans_map_tbl[i].transport_protocol) {
1164                         number_of_matches++;
1165                         dl.dl_match_fn(dl.dl_trans_map_tbl[i].network_protocol);
1166                         pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);
1167                         pf_compare_value(dl.dl_trans_map_tbl[i].offset +
1168                             dl.dl_link_header_len, 1,
1169                             transport_protocol);
1170                         pf_emit(ENF_AND);
1171                         if (number_of_matches > 1) {
1172                                 /*
1173                                  * Since we have two or more matches, in
1174                                  * order to have a correct and complete
1175                                  * program we need to OR the result of
1176                                  * each block of comparisons together.
1177                                  */
1178                                 pf_emit(ENF_OR);
1179                         }
1180                 }
1181         }
1182 }
1183
1184 static void
1185 pf_matchfn(const char *proto)
1186 {
1187         int i;
```

```
1189         for (i = 0; dl.dl_net_map_tbl[i].nmt_val != -1; i++) {
1190                 if (strcmp(proto, dl.dl_net_map_tbl[i].nmt_name) == 0) {
1191                         dl.dl_match_fn(dl.dl_net_map_tbl[i].nmt_val);
1192                         break;
1193                 }
1194         }
1195 }

1197 static void
1198 pf_primary()
1199 {
1200         for (;;) {
1201                 if (tokentype == FIELD)
1202                         break;

1204                 if (EQ("ip")) {
1205                         pf_matchfn("ip");
1206                         opstack++;
1207                         next();
1208                         break;
1209                 }

1211                 if (EQ("ip6")) {
1212                         pf_matchfn("ip6");
1213                         opstack++;
1214                         next();
1215                         break;
1216                 }

1218                 if (EQ("pppoe")) {
1219                         pf_matchfn("pppoe");
1220                         pf_match_ethertype(ETHERTYPE_PPPOES);
1221                         pf_emit(ENF_OR);
1222                         opstack++;
1223                         next();
1224                         break;
1225                 }

1227                 if (EQ("pppoed")) {
1228                         pf_matchfn("pppoed");
1229                         opstack++;
1230                         next();
1231                         break;
1232                 }

1234                 if (EQ("pppoes")) {
1235                         pf_matchfn("pppoes");
1236                         opstack++;
1237                         next();
1238                         break;
1239                 }

1241                 if (EQ("arp")) {
1242                         pf_matchfn("arp");
1243                         opstack++;
1244                         next();
1245                         break;
1246                 }

1248                 if (EQ("vlan")) {
1249                         pf_matchfn("vlan");
1250                         pf_compare_value_mask_neq(VLAN_ID_OFFSET, 2,
1251                             0, VLAN_ID_MASK);
1252                         pf_emit(ENF_AND);
1253                         opstack++;
1254                         next();
```



```
1255                         break;
1256                 }

1258                 if (EQ("vlan-id")) {
1259                         next();
1260                         if (tokentype != NUMBER)
1261                                 pr_err("VLAN ID expected");
1262                         pf_matchfn("vlan-id");
1263                         pf_compare_value_mask(VLAN_ID_OFFSET, 2, tokenval,
1264                             VLAN_ID_MASK);
1265                         pf_emit(ENF_AND);
1266                         opstack++;
1267                         next();
1268                         break;
1269                 }

1271                 if (EQ("rarp")) {
1272                         pf_matchfn("rarp");
1273                         opstack++;
1274                         next();
1275                         break;
1276                 }

1278                 if (EQ("tcp")) {
1279                         pf_check_transport_protocol(IPPROTO_TCP);
1280                         opstack++;
1281                         next();
1282                         break;
1283                 }

1285                 if (EQ("udp")) {
1286                         pf_check_transport_protocol(IPPROTO_UDP);
1287                         opstack++;
1288                         next();
1289                         break;
1290                 }

1292                 if (EQ("ospf")) {
1293                         pf_check_transport_protocol(IPPROTO_OSPF);
1294                         opstack++;
1295                         next();
1296                         break;
1297                 }


1300                 if (EQ("sctp")) {
1301                         pf_check_transport_protocol(IPPROTO_SCTP);
1302                         opstack++;
1303                         next();
1304                         break;
1305                 }

1307                 if (EQ("icmp")) {
1308                         pf_check_transport_protocol(IPPROTO_ICMP);
1309                         opstack++;
1310                         next();
1311                         break;
1312                 }

1314                 if (EQ("icmp6")) {
1315                         pf_check_transport_protocol(IPPROTO_ICMPV6);
1316                         opstack++;
1317                         next();
1318                         break;
1319                 }
```

```
1321                if (EQ("ip-in-ip")) {
1322                        pf_check_transport_protocol(IPPROTO_ENCAP);
1323                        opstack++;
1324                        next();
1325                        break;
1326                }

1328                if (EQ("esp")) {
1329                        pf_check_transport_protocol(IPPROTO_ESP);
1330                        opstack++;
1331                        next();
1332                        break;
1333                }

1335                if (EQ("ah")) {
1336                        pf_check_transport_protocol(IPPROTO_AH);
1337                        opstack++;
1338                        next();
1339                        break;
1340                }

1342                if (EQ("dccp")) {
1343                        pf_check_transport_protocol(IPPROTO_DCCP);
1344                        opstack++;
1345                        next();
1346                        break;
1347                }

1349 #endif /* ! codereview */
1350                if (EQ("(")) {
1351                        inBrace++;
1352                        next();
1353                        pf_expression();
1354                        if (EQ(")")) {
1355                                if (inBrace)
1356                                        inBraceOR--;
1357                                inBrace--;
1358                                next();
1359                        }
1360                        break;
1361                }

1363                if (EQ("to") || EQ("dst")) {
1364                        dir = TO;
1365                        next();
1366                        continue;
1367                }

1369                if (EQ("from") || EQ("src")) {
1370                        dir = FROM;
1371                        next();
1372                        continue;
1373                }

1375                if (EQ("ether")) {
1376                        eaddr = 1;
1377                        next();
1378                        continue;
1379                }

1381                if (EQ("inet")) {
1382                        next();
1383                        if (EQ("host"))
1384                                next();
1385                        if (tokentype != ALPHA && tokentype != ADDR_IP)
1386                                pr_err("host/IPv4 addr expected after inet");
```



```
1387                        pf_ipaddr_match(dir, token, IPV4_ONLY);
1388                        opstack++;
1389                        next();
1390                        break;
1391                }

1393                if (EQ("inet6")) {
1394                        next();
1395                        if (EQ("host"))
1396                                next();
1397                        if (tokentype != ALPHA && tokentype != ADDR_IP6)
1398                                pr_err("host/IPv6 addr expected after inet6");
1399                        pf_ipaddr_match(dir, token, IPV6_ONLY);
1400                        opstack++;
1401                        next();
1402                        break;
1403                }

1405                if (EQ("proto")) {
1406                        next();
1407                        if (tokentype != NUMBER)
1408                                pr_err("IP proto type expected");
1409                        pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);
1410                        pf_compare_value(
1411                            IPV4_TYPE_HEADER_OFFSET + dl.dl_link_header_len, 1,
1412                            tokenval);
1413                        opstack++;
1414                        next();
1415                        break;
1416                }

1418                if (EQ("broadcast")) {
1419                        pf_clear_offset_register();
1420                        pf_compare_value(dl.dl_link_dest_offset, 4, 0xffffffff);
1421                        opstack++;
1422                        next();
1423                        break;
1424                }

1426                if (EQ("multicast")) {
1427                        pf_clear_offset_register();
1428                        pf_compare_value_mask(
1429                            dl.dl_link_dest_offset, 1, 0x01, 0x01);
1430                        opstack++;
1431                        next();
1432                        break;
1433                }

1435                if (EQ("ethertype")) {
1436                        next();
1437                        if (tokentype != NUMBER)
1438                                pr_err("ether type expected");
1439                        pf_match_ethertype(tokenval);
1440                        opstack++;
1441                        next();
1442                        break;
1443                }

1445                if (EQ("net") || EQ("dstnet") || EQ("srcnet")) {
1446                        if (EQ("dstnet"))
1447                                dir = TO;
1448                        else if (EQ("srcnet"))
1449                                dir = FROM;
1450                        next();
1451                        pf_netaddr_match(dir, token);
1452                        dir = ANY;
```

```
1453                        opstack++;
1454                        next();
1455                        break;
1456                }

1458                if (EQ("zone")) {
1459                        next();
1460                        if (tokentype != NUMBER)
1461                                pr_err("zoneid expected after inet");
1462                        pf_match_zone(dir, BE_32((uint32_t)(tokenval)));
1463                        opstack++;
1464                        next();
1465                        break;
1466                }

1468                /*
1469                 * Give up on anything that's obviously
1470                 * not a primary.
1471                 */
1472                if (EQ("and") || EQ("or") ||
1473                    EQ("not") || EQ("decnet") || EQ("apple") ||
1474                    EQ("length") || EQ("less") || EQ("greater") ||
1475                    EQ("port") || EQ("srcport") || EQ("dstport") ||
1476                    EQ("rpc") || EQ("gateway") || EQ("nofrag") ||
1477                    EQ("bootp") || EQ("dhcp") || EQ("dhcp6") ||
1478                    EQ("slp") || EQ("ldap")) {
1479                        break;
1480                }

1482                if (EQ("host") || EQ("between") ||
1483                    tokentype == ALPHA || /* assume its a hostname */
1484                    tokentype == ADDR_IP ||
1485                    tokentype == ADDR_IP6 ||
1486                    tokentype == ADDR_ETHER) {
1487                        if (EQ("host") || EQ("between"))
1488                                next();
1489                        if (eaddr || tokentype == ADDR_ETHER) {
1490                                pf_etheraddr_match(dir, token);
1491                        } else if (tokentype == ALPHA) {
1492                                pf_ipaddr_match(dir, token, IPV4_AND_IPV6);
1493                        } else if (tokentype == ADDR_IP) {
1494                                pf_ipaddr_match(dir, token, IPV4_ONLY);
1495                        } else {
1496                                pf_ipaddr_match(dir, token, IPV6_ONLY);
1497                        }
1498                        dir = ANY;
1499                        eaddr = 0;
1500                        opstack++;
1501                        next();
1502                        break;
1503                }

1505                break;  /* unknown token */
1506        }
1507 }

1509 static void
1510 pf_alternation()
1511 {
1512         int s = opstack;

1514         pf_primary();
1515         for (;;) {
1516                 if (EQ("and"))
1517                         next();
1518                 pf_primary();
```



```
1519                 if (opstack != s + 2)
1520                         break;
1521                 pf_emit(ENF_AND);
1522                 opstack--;
1523         }
1524 }

1526 static void
1527 pf_expression()
1528 {
1529         pf_alternation();
1530         while (EQ("or") || EQ(",")) {
1531                 if (inBrace)
1532                         inBraceOR++;
1533                 else
1534                         foundOR++;
1535                 next();
1536                 pf_alternation();
1537                 pf_emit(ENF_OR);
1538                 opstack--;
1539         }
1540 }

1542 /*
1543  * Attempt to compile the expression
1544  * in the string "e".  If we can generate
1545  * pf code for it then return 1 - otherwise
1546  * return 0 and leave it up to the user-level
1547  * filter.
1548  */
1549 int
1550 pf_compile(e, print)
1551         char *e;
1552         int print;
1553 {
1554         char *argstr;
1555         char *sav_str, *ptr, *sav_ptr;
1556         int inBr = 0, aheadOR = 0;

1558         argstr = strdup(e);
1559         sav_str = e;
1560         tkp = argstr;
1561         dir = ANY;

1563         pfp = &pf.Pf_Filter[0];
1564         if (setjmp(env)) {
1565                 return (0);
1566         }

1568         /*
1569          * Set media specific packet offsets that this code uses.
1570          */
1571         if (interface->mac_type == DL_ETHER) {
1572                 dl.dl_type = DL_ETHER;
1573                 dl.dl_match_fn = pf_match_ethertype;
1574                 dl.dl_trans_map_tbl = ether_transport_mapping_table;
1575                 dl.dl_net_map_tbl = ether_network_mapping_table;
1576                 dl.dl_link_header_len = 14;
1577                 dl.dl_link_type_offset = 12;
1578                 dl.dl_link_dest_offset = 0;
1579                 dl.dl_link_src_offset = 6;
1580                 dl.dl_link_addr_len = 6;
1581         }

1583         if (interface->mac_type == DL_IB) {
1584                 dl.dl_type = DL_IB;
```

```
1585                dl.dl_link_header_len = 4;
1586                dl.dl_link_type_offset = 0;
1587                dl.dl_link_dest_offset = dl.dl_link_src_offset = -1;
1588                dl.dl_link_addr_len = 20;
1589                dl.dl_match_fn = pf_match_ibtype;
1590                dl.dl_trans_map_tbl = ib_transport_mapping_table;
1591                dl.dl_net_map_tbl = ib_network_mapping_table;
1592        }

1594        if (interface->mac_type == DL_IPNET) {
1595                dl.dl_type = DL_IPNET;
1596                dl.dl_link_header_len = 24;
1597                dl.dl_link_type_offset = 0;
1598                dl.dl_link_dest_offset = dl.dl_link_src_offset = -1;
1599                dl.dl_link_addr_len = -1;
1600                dl.dl_match_fn = pf_match_ipnettype;
1601                dl.dl_trans_map_tbl = ipnet_transport_mapping_table;
1602                dl.dl_net_map_tbl = ipnet_network_mapping_table;
1603        }

1605        next();
1606        pf_expression();

1608        if (tokentype != EOL) {
1609                /*
1610                 * The idea here is to do as much filtering as possible in
1611                 * the kernel. So even if we find a token we don't understand,
1612                 * we try to see if we can still set up a portion of the filter
1613                 * in the kernel and use the userland filter to filter the
1614                 * remaining stuff. Obviously, if our filter expression is of
1615                 * type A AND B, we can filter A in kernel and then apply B
1616                 * to the packets that got through. The same is not true for
1617                 * a filter of type A OR B. We can't apply A first and then B
1618                 * on the packets filtered through A.
1619                 *
1620                 * (We need to keep track of the fact when we find an OR,
1621                 * and the fact that we are inside brackets when we find OR.
1622                 * The variable 'foundOR' tells us if there was an OR behind,
1623                 * 'inBraceOR' tells us if we found an OR before we could find
1624                 * the end brace i.e. ')', and variable 'aheadOR' checks if
1625                 * there is an OR in the expression ahead. if either of these
1626                 * cases become true, we can't split the filtering)
1627                 */

1629                if (foundOR || inBraceOR) {
1630                        /* FORGET IN KERNEL FILTERING */
1631                        return (0);
1632                } else {

1634                        /* CHECK IF NO OR AHEAD */
1635                        sav_ptr = (char *)((uintptr_t)sav_str +
1636                                           (uintptr_t)sav_tkp -
1637                                           (uintptr_t)argstr);
1638                        ptr = sav_ptr;
1639                        while (*ptr != '\0') {
1640                                switch (*ptr) {
1641                                case '(':
1642                                        inBr++;
1643                                        break;
1644                                case ')':
1645                                        inBr--;
1646                                        break;
1647                                case 'o':
1648                                case 'O':
1649                                        if ((*(ptr + 1) == 'R' ||
1650                                             *(ptr + 1) == 'r') && !inBr)
```



```
1651                                                aheadOR = 1;
1652                                        break;
1653                                case ',':
1654                                        if (!inBr)
1655                                                aheadOR = 1;
1656                                        break;
1657                                }
1658                                ptr++;
1659                        }
1660                        if (!aheadOR) {
1661                                /* NO OR AHEAD, SPLIT UP THE FILTERING */
1662                                pf.Pf_FilterLen = pfp - &pf.Pf_Filter[0];
1663                                pf.Pf_Priority = 5;
1664                                if (print) {
1665                                        pf_codeprint(&pf.Pf_Filter[0],
1666                                                     pf.Pf_FilterLen);
1667                                }
1668                                compile(sav_ptr, print);
1669                                return (2);
1670                        } else
1671                                return (0);
1672                }
1673        }

1675        pf.Pf_FilterLen = pfp - &pf.Pf_Filter[0];
1676        pf.Pf_Priority = 5;     /* unimportant, so long as > 2 */
1677        if (print) {
1678                pf_codeprint(&pf.Pf_Filter[0], pf.Pf_FilterLen);
1679        }
1680        return (1);
1681 }
```

****
10831 Mon Jul 9 14:38:08 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_rport.c
dccp: options and features
****
_____***unchanged_portion_omitted_***

```
 140 char *
 141 getportname(int proto, in_port_t port)
 142 {
 143         const struct porttable *p, *pt;

 145         switch (proto) {
 146         case IPPROTO_DCCP: /* fallthru */
 147         case IPPROTO_SCTP:
 146         case IPPROTO_SCTP: /* fallthru */
 148         case IPPROTO_TCP: pt = pt_tcp; break;
 149         case IPPROTO_UDP: pt = pt_udp; break;
 150         default: return (NULL);
 151         }

 153         for (p = pt; p->pt_num; p++) {
 154                 if (port == p->pt_num)
 155                         return (p->pt_short);
 156         }
 157         return (NULL);
 158 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18154 Mon Jul  9 14:38:08 2012
new/usr/src/cmd/devfsadm/misc_link.c
dccp: fix setsockopt bug
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1998, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

#include <regex.h>
#include <devfsadm.h>
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>
#include <limits.h>
#include <sys/zone.h>
#include <sys/zcons.h>
#include <sys/cpuid_drv.h>

static int display(di_minor_t minor, di_node_t node);
static int parallel(di_minor_t minor, di_node_t node);
static int node_slash_minor(di_minor_t minor, di_node_t node);
static int driver_minor(di_minor_t minor, di_node_t node);
static int node_name(di_minor_t minor, di_node_t node);
static int minor_name(di_minor_t minor, di_node_t node);
static int wifi_minor_name(di_minor_t minor, di_node_t node);
static int conskbd(di_minor_t minor, di_node_t node);
static int consms(di_minor_t minor, di_node_t node);
static int power_button(di_minor_t minor, di_node_t node);
static int fc_port(di_minor_t minor, di_node_t node);
static int printer_create(di_minor_t minor, di_node_t node);
static int se_hdlc_create(di_minor_t minor, di_node_t node);
static int ppm(di_minor_t minor, di_node_t node);
static int gpio(di_minor_t minor, di_node_t node);
static int av_create(di_minor_t minor, di_node_t node);
static int tsalarm_create(di_minor_t minor, di_node_t node);
static int ntwdt_create(di_minor_t minor, di_node_t node);
static int zcons_create(di_minor_t minor, di_node_t node);
static int cpuid(di_minor_t minor, di_node_t node);
static int glvc(di_minor_t minor, di_node_t node);
static int ses_callback(di_minor_t minor, di_node_t node);
static int kmdrv_create(di_minor_t minor, di_node_t node);

static devfsadm_create_t misc_cbt[] = {
        { "pseudo", "ddi_pseudo", "(^sad$)",
```



```
            TYPE_EXACT | DRV_RE, ILEVEL_0, node_slash_minor
        },
        { "pseudo", "ddi_pseudo", "zsh",
            TYPE_EXACT | DRV_EXACT, ILEVEL_0, driver_minor
        },
        { "network", "ddi_network", NULL,
            TYPE_EXACT, ILEVEL_0, minor_name
        },
        { "wifi", "ddi_network:wifi", NULL,
            TYPE_EXACT, ILEVEL_0, wifi_minor_name
        },
        { "display", "ddi_display", NULL,
            TYPE_EXACT, ILEVEL_0, display
        },
        { "parallel", "ddi_parallel", NULL,
            TYPE_EXACT, ILEVEL_0, parallel
        },
        { "enclosure", DDI_NT_SCSI_ENCLOSURE, NULL,
            TYPE_EXACT, ILEVEL_0, ses_callback
        },
        { "pseudo", "ddi_pseudo", "(^winlock$)|(^pm$)",
            TYPE_EXACT | DRV_RE, ILEVEL_0, node_name
        },
        { "pseudo", "ddi_pseudo", "conskbd",
            TYPE_EXACT | DRV_EXACT, ILEVEL_0, conskbd
        },
        { "pseudo", "ddi_pseudo", "consms",
            TYPE_EXACT | DRV_EXACT, ILEVEL_0, consms
        },
        { "pseudo", "ddi_pseudo", "rsm",
            TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name
        },
        { "pseudo", "ddi_pseudo",
            "(^lockstat$)|(^SUNW,rtvc$)|(^vol$)|(^log$)|(^sy$)|"
            "(^ksyms$)|(^clone$)|(^tl$)|(^tnf$)|(^kstat$)|(^mdesc$)|(^eeprom$)|"
            "(^ptsl$)|(^mm$)|(^wc$)|(^dump$)|(^cn$)|(^svvslo$)|(^ptm$)|"
            "(^ptc$)|(^openeepr$)|(^poll$)|(^sysmsg$)|(^random$)|(^trapstat$)|"
            "(^cryptoadm$)|(^crypto$)|(^pool$)|(^poolctl$)|(^bl$)|(^kmdb$)|"
            "(^sysevent$)|(^kssl$)|(^physmem$)",
            TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name
        },
        { "pseudo", "ddi_pseudo",
            "(^ip$)|(^tcp$)|(^udp$)|(^icmp$)|(^dccp$)|"
            "(^ip6$)|(^tcp6$)|(^udp6$)|(^icmp6$)|(^dccp6$)|"
            "(^ip$)|(^tcp$)|(^udp$)|(^icmp$)|"
            "(^ip6$)|(^tcp6$)|(^udp6$)|(^icmp6$)|"
            "(^rts$)|(^arp$)|(^ipsecah$)|(^ipsecesp$)|(^keysock$)|(^spdsock$)|"
            "(^nca$)|(^rds$)|(^sdp$)|(^ipnet$)|(^dlpistub$)|(^bpf$)",
            TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name
        },
        { "pseudo", "ddi_pseudo",
            "(^ipf$)|(^ipnat$)|(^ipstate$)|(^ipauth$)|"
            "(^ipsync$)|(^ipscan$)|(^iplookup$)",
            TYPE_EXACT | DRV_RE, ILEVEL_0, minor_name,
        },
        { "pseudo", "ddi_pseudo", "dld",
            TYPE_EXACT | DRV_EXACT, ILEVEL_0, node_name
        },
        { "pseudo", "ddi_pseudo",
            "(^kdmouse$)|(^rootprop$)",
            TYPE_EXACT | DRV_RE, ILEVEL_0, node_name
        },
        { "pseudo", "ddi_pseudo", "tod",
            TYPE_EXACT | DRV_EXACT, ILEVEL_0, node_name
        },
        { "pseudo", "ddi_pseudo", "envctrl(two)?",
```

```
126             TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name,
127         },
128         { "pseudo", "ddi_pseudo", "fcode",
129             TYPE_EXACT | DRV_RE, ILEVEL_0, minor_name,
130         },
131         { "power_button", "ddi_power_button", NULL,
132             TYPE_EXACT, ILEVEL_0, power_button,
133         },
134         { "FC port", "ddi_ctl:devctl", "fp",
135             TYPE_EXACT | DRV_EXACT, ILEVEL_0, fc_port
136         },
137         { "printer", "ddi_printer", NULL,
138             TYPE_EXACT, ILEVEL_0, printer_create
139         },
140         { "pseudo", "ddi_pseudo", "se",
141             TYPE_EXACT | DRV_EXACT, ILEVEL_0, se_hdlc_create
142         },
143         { "ppm",  "ddi_ppm", NULL,
144             TYPE_EXACT, ILEVEL_0, ppm
145         },
146         { "pseudo", "ddi_pseudo", "gpio_87317",
147             TYPE_EXACT | DRV_EXACT, ILEVEL_0, gpio
148         },
149         { "pseudo", "ddi_pseudo", "sckmdrv",
150             TYPE_EXACT | DRV_RE, ILEVEL_0, kmdrv_create,
151         },
152         { "pseudo", "ddi_pseudo", "oplkmdrv",
153             TYPE_EXACT | DRV_RE, ILEVEL_0, kmdrv_create,
154         },
155         { "av", "^ddi_av:(isoch|async)$", NULL,
156             TYPE_RE, ILEVEL_0, av_create,
157         },
158         { "pseudo", "ddi_pseudo", "tsalarm",
159             TYPE_EXACT | DRV_RE, ILEVEL_0, tsalarm_create,
160         },
161         { "pseudo", "ddi_pseudo", "ntwdt",
162             TYPE_EXACT | DRV_RE, ILEVEL_0, ntwdt_create,
163         },
164         { "pseudo", "ddi_pseudo", "daplt",
165             TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name
166         },
167         { "pseudo", "ddi_pseudo", "zcons",
168             TYPE_EXACT | DRV_EXACT, ILEVEL_0, zcons_create,
169         },
170         { "pseudo", "ddi_pseudo", CPUID_DRIVER_NAME,
171             TYPE_EXACT | DRV_EXACT, ILEVEL_0, cpuid,
172         },
173         { "pseudo", "ddi_pseudo", "glvc",
174             TYPE_EXACT | DRV_EXACT, ILEVEL_0, glvc,
175         },
176         { "pseudo", "ddi_pseudo", "dm2s",
177             TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name,
178         },
179         { "pseudo", "ddi_pseudo", "nsmb",
180             TYPE_EXACT | DRV_EXACT, ILEVEL_1, minor_name,
181         },
182         { "pseudo", "ddi_pseudo", "mem_cache",
183             TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name,
184         },
185         { "pseudo", "ddi_pseudo", "fm",
186             TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name,
187         },
188         { "pseudo", "ddi_pseudo", "smbsrv",
189             TYPE_EXACT | DRV_EXACT, ILEVEL_1, minor_name,
190         },
191         { "pseudo", "ddi_pseudo", "tpm",
```



```
192             TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name
193         },
194 };
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  128972 Mon Jul  9 14:38:09 2012
new/usr/src/cmd/mdb/common/modules/genunix/genunix.c
dccp: build fixes, mdb (vfs sonode missing)
**********************************************************
______unchanged_portion_omitted_

3841 static const mdb_dcmd_t dcmds[] = {

3843         /* from genunix.c */
3844         { "as2proc", ":", "convert as to proc_t address", as2proc },
3845         { "binding_hash_entry", ":", "print driver names hash table entry",
3846                 binding_hash_entry },
3847         { "callout", "?[-r|n] [-s|l] [-xhB] [-t | -ab nsec [-dkD]]"
3848             " [-C addr | -S seqid] [-f name|addr] [-p name| addr] [-T|L [-E]]"
3849             " [-FivVA]",
3850             "display callouts", callout, callout_help },
3851         { "calloutid", "[-d|v] xid", "print callout by extended id",
3852             calloutid, calloutid_help },
3853         { "class", NULL, "print process scheduler classes", class },
3854         { "cpuinfo", "?[-v]", "print CPUs and runnable threads", cpuinfo },
3855         { "did2thread", "? kt_did", "find kernel thread for this id",
3856                 did2thread },
3857         { "errorq", "?[-v]", "display kernel error queues", errorq },
3858         { "fd", ":[fd num]", "get a file pointer from an fd", fd },
3859         { "flipone", ":", "the vik_rev_level 2 special", flipone },
3860         { "lminfo", NULL, "print lock manager information", lminfo },
3861         { "ndi_event_hdl", "?", "print ndi_event_hdl", ndi_event_hdl },
3862         { "panicinfo", NULL, "print panic information", panicinfo },
3863         { "pid2proc", "?", "convert PID to proc_t address", pid2proc },
3864         { "project", NULL, "display kernel project(s)", project },
3865         { "ps", "[-fltzTP]", "list processes (and associated thr,lwp)", ps },
3866         { "pgrep", "[-x] [-n | -o] pattern",
3867                 "pattern match against all processes", pgrep },
3868         { "ptree", NULL, "print process tree", ptree },
3869         { "sysevent", "?[-sv]", "print sysevent pending or sent queue",
3870                 sysevent},
3871         { "sysevent_channel", "?", "print sysevent channel database",
3872                 sysevent_channel},
3873         { "sysevent_class_list", ":", "print sysevent class list",
3874                 sysevent_class_list},
3875         { "sysevent_subclass_list", ":",
3876                 "print sysevent subclass list", sysevent_subclass_list},
3877         { "system", NULL, "print contents of /etc/system file", sysfile },
3878         { "task", NULL, "display kernel task(s)", task },
3879         { "time", "[-dlx]", "display system time", time, time_help },
3880         { "vnode2path", ":[-F]", "vnode address to pathname", vnode2path },
3881         { "whereopen", ":", "given a vnode, dumps procs which have it open",
3882             whereopen },

3884         /* from bio.c */
3885         { "bufpagefind", ":addr", "find page_t on buf_t list", bufpagefind },

3887         /* from bitset.c */
3888         { "bitset", ":", "display a bitset", bitset, bitset_help },

3890         /* from contract.c */
3891         { "contract", "?", "display a contract", cmd_contract },
3892         { "ctevent", ":", "display a contract event", cmd_ctevent },
3893         { "ctid", ":", "convert id to a contract pointer", cmd_ctid },

3895         /* from cpupart.c */
3896         { "cpupart", "?[-v]", "print cpu partition info", cpupart },

3898         /* from cred.c */
3899         { "cred", ":[-v]", "display a credential", cmd_cred },
```



```
3900         { "credgrp", ":[-v]", "display cred_t groups", cmd_credgrp },
3901         { "credsid", ":[-v]", "display a credsid_t", cmd_credsid },
3902         { "ksidlist", ":[-v]", "display a ksidlist_t", cmd_ksidlist },

3904         /* from cyclic.c */
3905         { "cyccover", NULL, "dump cyclic coverage information", cyccover },
3906         { "cycid", "?", "dump a cyclic id", cycid },
3907         { "cycinfo", "?", "dump cyc_cpu info", cycinfo },
3908         { "cyclic", ":", "developer information", cyclic },
3909         { "cyctrace", "?", "dump cyclic trace buffer", cyctrace },

3911         /* from damap.c */
3912         { "damap", ":", "display a damap_t", damap, damap_help },

3914         /* from devinfo.c */
3915         { "devbindings", "?[-qs] [device-name | major-num]",
3916             "print devinfo nodes bound to device-name or major-num",
3917             devbindings, devinfo_help },
3918         { "devinfo", ":[-qs]", "detailed devinfo of one node", devinfo,
3919             devinfo_help },
3920         { "devinfo_audit", ":[-v]", "devinfo configuration audit record",
3921             devinfo_audit },
3922         { "devinfo_audit_log", "?[-v]", "system wide devinfo configuration log",
3923             devinfo_audit_log },
3924         { "devinfo_audit_node", ":[-v]", "devinfo node configuration history",
3925             devinfo_audit_node },
3926         { "devinfo2driver", ":", "find driver name for this devinfo node",
3927             devinfo2driver },
3928         { "devnames", "?[-vm] [num]", "print devnames array", devnames },
3929         { "dev2major", "?<dev_t>", "convert dev_t to a major number",
3930             dev2major },
3931         { "dev2minor", "?<dev_t>", "convert dev_t to a minor number",
3932             dev2minor },
3933         { "devt", "?<dev_t>", "display a dev_t's major and minor numbers",
3934             devt },
3935         { "major2name", "?<major-num>", "convert major number to dev name",
3936             major2name },
3937         { "minornodes", ":", "given a devinfo node, print its minor nodes",
3938             minornodes },
3939         { "modctl2devinfo", ":", "given a modctl, list its devinfos",
3940             modctl2devinfo },
3941         { "name2major", "<dev-name>", "convert dev name to major number",
3942             name2major },
3943         { "prtconf", "?[-vpc]", "print devinfo tree", prtconf, prtconf_help },
3944         { "softstate", ":<instance>", "retrieve soft-state pointer",
3945             softstate },
3946         { "devinfo_fm", ":", "devinfo fault managment configuration",
3947             devinfo_fm },
3948         { "devinfo_fmce", ":", "devinfo fault managment cache entry",
3949             devinfo_fmce},

3951         /* from findstack.c */
3952         { "findstack", ":[-v]", "find kernel thread stack", findstack },
3953         { "findstack_debug", NULL, "toggle findstack debugging",
3954                 findstack_debug },
3955         { "stacks", "?[-afiv] [-c func] [-C func] [-m module] [-M module] "
3956                 "[-s sobj | -S sobj] [-t tstate | -T tstate]",
3957                 "print unique kernel thread stacks",
3958                 stacks, stacks_help },

3960         /* from fm.c */
3961         { "ereport", "[-v]", "print ereports logged in dump",
3962             ereport },

3964         /* from group.c */
3965         { "group", "?[-q]", "display a group", group},
```

```
3967         /* from hotplug.c */
3968         { "hotplug", "?[-p]", "display a registered hotplug attachment",
3969             hotplug, hotplug_help },

3971         /* from irm.c */
3972         { "irmpools", NULL, "display interrupt pools", irmpools_dcmd },
3973         { "irmreqs", NULL, "display interrupt requests in an interrupt pool",
3974             irmreqs_dcmd },
3975         { "irmreq", NULL, "display an interrupt request", irmreq_dcmd },

3977         /* from kgrep.c + genunix.c */
3978         { "kgrep", KGREP_USAGE, "search kernel as for a pointer", kgrep,
3979               kgrep_help },

3981         /* from kmem.c */
3982         { "allocdby", ":", "given a thread, print its allocated buffers",
3983               allocdby },
3984         { "bufctl", ":[-vh] [-a addr] [-c caller] [-e earliest] [-l latest] "
3985               "[-t thd]", "print or filter a bufctl", bufctl, bufctl_help },
3986         { "freedby", ":", "given a thread, print its freed buffers", freedby },
3987         { "kmalog", "?[ fail | slab ]",
3988             "display kmem transaction log and stack traces", kmalog },
3989         { "kmastat", "[-kmg]", "kernel memory allocator stats",
3990             kmastat },
3991         { "kmausers", "?[-ef] [cache ...]", "current medium and large users "
3992               "of the kmem allocator", kmausers, kmausers_help },
3993         { "kmem_cache", "?[-n name]",
3994               "print kernel memory caches", kmem_cache, kmem_cache_help},
3995         { "kmem_slabs", "?[-v] [-n cache] [-N cache] [-b maxbins] "
3996               "[-B minbinsize]", "display slab usage per kmem cache",
3997               kmem_slabs, kmem_slabs_help },
3998         { "kmem_debug", NULL, "toggle kmem dcmd/walk debugging", kmem_debug },
3999         { "kmem_log", "?[-b]", "dump kmem transaction log", kmem_log },
4000         { "kmem_verify", "?", "check integrity of kmem-managed memory",
4001               kmem_verify },
4002         { "vmem", "?", "print a vmem_t", vmem },
4003         { "vmem_seg", ":[-sv] [-c caller] [-e earliest] [-l latest] "
4004               "[-m minsize] [-M maxsize] [-t thread] [-T type]",
4005               "print or filter a vmem_seg", vmem_seg, vmem_seg_help },
4006         { "whatthread", ":[-v]", "print threads whose stack contains the "
4007               "given address", whatthread },

4009         /* from ldi.c */
4010         { "ldi_handle", "?[-i]", "display a layered driver handle",
4011             ldi_handle, ldi_handle_help },
4012         { "ldi_ident", NULL, "display a layered driver identifier",
4013             ldi_ident, ldi_ident_help },

4015         /* from leaky.c + leaky_subr.c */
4016         { "findleaks", FINDLEAKS_USAGE,
4017             "search for potential kernel memory leaks", findleaks,
4018             findleaks_help },

4020         /* from lgrp.c */
4021         { "lgrp", "?[-q] [-p | -Pih]", "display an lgrp", lgrp},
4022         { "lgrp_set", "", "display bitmask of lgroups as a list", lgrp_set},

4024         /* from log.c */
4025         { "msgbuf", "?[-v]", "print most recent console messages", msgbuf },

4027         /* from mdi.c */
4028         { "mdipi", NULL, "given a path, dump mdi_pathinfo "
4029               "and detailed pi_prop list", mdipi },
4030         { "mdiprops", NULL, "given a pi_prop, dump the pi_prop list",
4031               mdiprops },
```



```
4032         { "mdiphci", NULL, "given a phci, dump mdi_phci and "
4033               "list all paths", mdiphci },
4034         { "mdivhci", NULL, "given a vhci, dump mdi_vhci and list "
4035               "all phcis", mdivhci },
4036         { "mdiclient_paths", NULL, "given a path, walk mdi_pathinfo "
4037               "client links", mdiclient_paths },
4038         { "mdiphci_paths", NULL, "given a path, walk through mdi_pathinfo "
4039               "phci links", mdiphci_paths },
4040         { "mdiphcis", NULL, "given a phci, walk through mdi_phci ph_next links",
4041               mdiphcis },

4043         /* from memory.c */
4044         { "addr2smap", ":[offset]", "translate address to smap", addr2smap },
4045         { "memlist", "?[-iav]", "display a struct memlist", memlist },
4046         { "memstat", NULL, "display memory usage summary", memstat },
4047         { "page", "?", "display a summarized page_t", page },
4048         { "pagelookup", "?[-v vp] [-o offset]",
4049               "find the page_t with the name {vp, offset}",
4050               pagelookup, pagelookup_help },
4051         { "page_num2pp", ":", "find the page_t for a given page frame number",
4052               page_num2pp },
4053         { "pmap", ":[-q]", "print process memory map", pmap },
4054         { "seg", ":", "print address space segment", seg },
4055         { "swapinfo", "?", "display a struct swapinfo", swapinfof },
4056         { "vnode2smap", ":[offset]", "translate vnode to smap", vnode2smap },

4058         /* from mmd.c */
4059         { "multidata", ":[-sv]", "display a summarized multidata_t",
4060               multidata },
4061         { "pattbl", ":", "display a summarized multidata attribute table",
4062               pattbl },
4063         { "pattr2multidata", ":", "print multidata pointer from pattr_t",
4064               pattr2multidata },
4065         { "pdesc2slab", ":", "print pdesc slab pointer from pdesc_t",
4066               pdesc2slab },
4067         { "pdesc_verify", ":", "verify integrity of a pdesc_t", pdesc_verify },
4068         { "slab2multidata", ":", "print multidata pointer from pdesc_slab_t",
4069               slab2multidata },

4071         /* from modhash.c */
4072         { "modhash", "?[-ceht] [-k key] [-v val] [-i index]",
4073               "display information about one or all mod_hash structures",
4074               modhash, modhash_help },
4075         { "modent", ":[-k | -v | -t type]",
4076               "display information about a mod_hash_entry", modent,
4077               modent_help },

4079         /* from net.c */
4080         { "dladm", "?<sub-command> [flags]", "show data link information",
4081               dladm, dladm_help },
4082         { "mi", ":[-p] [-d | -m]", "filter and display MI object or payload",
4083               mi },
4084         { "netstat", "[-arv] [-f inet | inet6 | unix] [-P tcp | udp | icmp | dcc
4084         { "netstat", "[-arv] [-f inet | inet6 | unix] [-P tcp | udp | icmp]",
4085               "show network statistics", netstat },
4086         { "sonode", "?[-f inet | inet6 | unix | #] "
4087               "[-t stream | dgram | raw | #] [-p #]",
4088               "filter and display sonode", sonode },

4090         /* from netstack.c */
4091         { "netstack", "", "show stack instances", netstack },

4093         /* from nvpair.c */
4094         { NVPAIR_DCMD_NAME, NVPAIR_DCMD_USAGE, NVPAIR_DCMD_DESCR,
4095               nvpair_print },
4096         { NVLIST_DCMD_NAME, NVLIST_DCMD_USAGE, NVLIST_DCMD_DESCR,
```

```
4097                 print_nvlist },

4099         /* from pg.c */
4100         { "pg", "?[-q]", "display a pg", pg},

4102         /* from rctl.c */
4103         { "rctl_dict", "?", "print systemwide default rctl definitions",
4104                 rctl_dict },
4105         { "rctl_list", ":[handle]", "print rctls for the given proc",
4106                 rctl_list },
4107         { "rctl", ":[handle]", "print a rctl_t, only if it matches the handle",
4108                 rctl },
4109         { "rctl_validate", ":[-v] [-n #]", "test resource control value "
4110                 "sequence", rctl_validate },

4112         /* from sobj.c */
4113         { "rwlock", ":", "dump out a readers/writer lock", rwlock },
4114         { "mutex", ":[-f]", "dump out an adaptive or spin mutex", mutex,
4115                 mutex_help },
4116         { "sobj2ts", ":", "perform turnstile lookup on synch object", sobj2ts },
4117         { "wchaninfo", "?[-v]", "dump condition variable", wchaninfo },
4118         { "turnstile", "?", "display a turnstile", turnstile },

4120         /* from stream.c */
4121         { "mblk", ":[-q|v] [-f|F flag] [-t|T type] [-l|L|B len] [-d dbaddr]",
4122                 "print an mblk", mblk_prt, mblk_help },
4123         { "mblk_verify", "?", "verify integrity of an mblk", mblk_verify },
4124         { "mblk2dblk", ":", "convert mblk_t address to dblk_t address",
4125                 mblk2dblk },
4126         { "q2otherq", ":", "print peer queue for a given queue", q2otherq },
4127         { "q2rdq", ":", "print read queue for a given queue", q2rdq },
4128         { "q2syncq", ":", "print syncq for a given queue", q2syncq },
4129         { "q2stream", ":", "print stream pointer for a given queue", q2stream },
4130         { "q2wrq", ":", "print write queue for a given queue", q2wrq },
4131         { "queue", ":[-q|v] [-m mod] [-f flag] [-F flag] [-s syncq_addr]",
4132                 "filter and display STREAM queue", queue, queue_help },
4133         { "stdata", ":[-q|v] [-f flag] [-F flag]",
4134                 "filter and display STREAM head", stdata, stdata_help },
4135         { "str2mate", ":", "print mate of this stream", str2mate },
4136         { "str2wrq", ":", "print write queue of this stream", str2wrq },
4137         { "stream", ":", "display STREAM", stream },
4138         { "strftevent", ":", "print STREAMS flow trace event", strftevent },
4139         { "syncq", ":[-q|v] [-f flag] [-F flag] [-t type] [-T type]",
4140                 "filter and display STREAM sync queue", syncq, syncq_help },
4141         { "syncq2q", ":", "print queue for a given syncq", syncq2q },

4143         /* from taskq.c */
4144         { "taskq", ":[-atT] [-m min_maxq] [-n name]",
4145             "display a taskq", taskq, taskq_help },
4146         { "taskq_entry", ":", "display a taskq_ent_t", taskq_ent },

4148         /* from thread.c */
4149         { "thread", "?[-bdfimps]", "display a summarized kthread_t", thread,
4150                 thread_help },
4151         { "threadlist", "?[-t] [-v [count]]",
4152                 "display threads and associated C stack traces", threadlist,
4153                 threadlist_help },
4154         { "stackinfo", "?[-h|-a]", "display kthread_t stack usage", stackinfo,
4155                 stackinfo_help },

4157         /* from tsd.c */
4158         { "tsd", ":-k key", "print tsd[key-1] for this thread", ttotsd },
4159         { "tsdtot", ":", "find thread with this tsd", tsdtot },

4161         /*
4162          * typegraph does not work under kmdb, as it requires too much memory
```



```
4163          * for its internal data structures.
4164          */
4165 #ifndef _KMDB
4166         /* from typegraph.c */
4167         { "findlocks", ":", "find locks held by specified thread", findlocks },
4168         { "findfalse", "?[-v]", "find potentially falsely shared structures",
4169                 findfalse },
4170         { "typegraph", NULL, "build type graph", typegraph },
4171         { "istype", ":type", "manually set object type", istype },
4172         { "notype", ":", "manually clear object type", notype },
4173         { "whattype", ":", "determine object type", whattype },
4174 #endif

4176         /* from vfs.c */
4177         { "fsinfo", "?[-v]", "print mounted filesystems", fsinfo },
4178         { "pfiles", ":[-fp]", "print process file information", pfiles,
4179                 pfiles_help },

4181         /* from zone.c */
4182         { "zone", "?[-r [-v]]", "display kernel zone(s)", zoneprt },
4183         { "zsd", ":[-v] [zsd_key]", "display zone-specific-data entries for "
4184             "selected zones", zsd },

4186         { NULL }
4187 };

4189 static const mdb_walker_t walkers[] = {

4191         /* from genunix.c */
4192         { "callouts_bytime", "walk callouts by list chain (expiration time)",
4193                 callout_walk_init, callout_walk_step, callout_walk_fini,
4194                 (void *)CALLOUT_WALK_BYLIST },
4195         { "callouts_byid", "walk callouts by id hash chain",
4196                 callout_walk_init, callout_walk_step, callout_walk_fini,
4197                 (void *)CALLOUT_WALK_BYID },
4198         { "callout_list", "walk a callout list", callout_list_walk_init,
4199                 callout_list_walk_step, callout_list_walk_fini },
4200         { "callout_table", "walk callout table array", callout_table_walk_init,
4201                 callout_table_walk_step, callout_table_walk_fini },
4202         { "cpu", "walk cpu structures", cpu_walk_init, cpu_walk_step },
4203         { "ereportq_dump", "walk list of ereports in dump error queue",
4204                 ereportq_dump_walk_init, ereportq_dump_walk_step, NULL },
4205         { "ereportq_pend", "walk list of ereports in pending error queue",
4206                 ereportq_pend_walk_init, ereportq_pend_walk_step, NULL },
4207         { "errorq", "walk list of system error queues",
4208                 errorq_walk_init, errorq_walk_step, NULL },
4209         { "errorq_data", "walk pending error queue data buffers",
4210                 eqd_walk_init, eqd_walk_step, eqd_walk_fini },
4211         { "allfile", "given a proc pointer, list all file pointers",
4212                 file_walk_init, allfile_walk_step, file_walk_fini },
4213         { "file", "given a proc pointer, list of open file pointers",
4214                 file_walk_init, file_walk_step, file_walk_fini },
4215         { "lock_descriptor", "walk lock_descriptor_t structures",
4216                 ld_walk_init, ld_walk_step, NULL },
4217         { "lock_graph", "walk lock graph",
4218                 lg_walk_init, lg_walk_step, NULL },
4219         { "port", "given a proc pointer, list of created event ports",
4220                 port_walk_init, port_walk_step, NULL },
4221         { "portev", "given a port pointer, list of events in the queue",
4222                 portev_walk_init, portev_walk_step, portev_walk_fini },
4223         { "proc", "list of active proc_t structures",
4224                 proc_walk_init, proc_walk_step, proc_walk_fini },
4225         { "projects", "walk a list of kernel projects",
4226                 project_walk_init, project_walk_step, NULL },
4227         { "sysevent_pend", "walk sysevent pending queue",
4228                 sysevent_pend_walk_init, sysevent_walk_step,
```

```
4229                 sysevent_walk_fini},
4230         { "sysevent_sent", "walk sysevent sent queue", sysevent_sent_walk_init,
4231                 sysevent_walk_step, sysevent_walk_fini},
4232         { "sysevent_channel", "walk sysevent channel subscriptions",
4233                 sysevent_channel_walk_init, sysevent_channel_walk_step,
4234                 sysevent_channel_walk_fini},
4235         { "sysevent_class_list", "walk sysevent subscription's class list",
4236                 sysevent_class_list_walk_init, sysevent_class_list_walk_step,
4237                 sysevent_class_list_walk_fini},
4238         { "sysevent_subclass_list",
4239                 "walk sysevent subscription's subclass list",
4240                 sysevent_subclass_list_walk_init,
4241                 sysevent_subclass_list_walk_step,
4242                 sysevent_subclass_list_walk_fini},
4243         { "task", "given a task pointer, walk its processes",
4244                 task_walk_init, task_walk_step, NULL },

4246         /* from avl.c */
4247         { AVL_WALK_NAME, AVL_WALK_DESC,
4248                 avl_walk_init, avl_walk_step, avl_walk_fini },

4250         /* from bio.c */
4251         { "buf", "walk the bio buf hash",
4252                 buf_walk_init, buf_walk_step, buf_walk_fini },

4254         /* from contract.c */
4255         { "contract", "walk all contracts, or those of the specified type",
4256                 ct_walk_init, generic_walk_step, NULL },
4257         { "ct_event", "walk events on a contract event queue",
4258                 ct_event_walk_init, generic_walk_step, NULL },
4259         { "ct_listener", "walk contract event queue listeners",
4260                 ct_listener_walk_init, generic_walk_step, NULL },

4262         /* from cpupart.c */
4263         { "cpupart_cpulist", "given an cpupart_t, walk cpus in partition",
4264                 cpupart_cpulist_walk_init, cpupart_cpulist_walk_step,
4265                 NULL },
4266         { "cpupart_walk", "walk the set of cpu partitions",
4267                 cpupart_walk_init, cpupart_walk_step, NULL },

4269         /* from ctxop.c */
4270         { "ctxop", "walk list of context ops on a thread",
4271                 ctxop_walk_init, ctxop_walk_step, ctxop_walk_fini },

4273         /* from cyclic.c */
4274         { "cyccpu", "walk per-CPU cyc_cpu structures",
4275                 cyccpu_walk_init, cyccpu_walk_step, NULL },
4276         { "cycomni", "for an omnipresent cyclic, walk cyc_omni_cpu list",
4277                 cycomni_walk_init, cycomni_walk_step, NULL },
4278         { "cyctrace", "walk cyclic trace buffer",
4279                 cyctrace_walk_init, cyctrace_walk_step, cyctrace_walk_fini },

4281         /* from devinfo.c */
4282         { "binding_hash", "walk all entries in binding hash table",
4283                 binding_hash_walk_init, binding_hash_walk_step, NULL },
4284         { "devinfo", "walk devinfo tree or subtree",
4285                 devinfo_walk_init, devinfo_walk_step, devinfo_walk_fini },
4286         { "devinfo_audit_log", "walk devinfo audit system-wide log",
4287                 devinfo_audit_log_walk_init, devinfo_audit_log_walk_step,
4288                 devinfo_audit_log_walk_fini},
4289         { "devinfo_audit_node", "walk per-devinfo audit history",
4290                 devinfo_audit_node_walk_init, devinfo_audit_node_walk_step,
4291                 devinfo_audit_node_walk_fini},
4292         { "devinfo_children", "walk children of devinfo node",
4293                 devinfo_children_walk_init, devinfo_children_walk_step,
4294                 devinfo_children_walk_fini },
```



```
4295         { "devinfo_parents", "walk ancestors of devinfo node",
4296                 devinfo_parents_walk_init, devinfo_parents_walk_step,
4297                 devinfo_parents_walk_fini },
4298         { "devinfo_siblings", "walk siblings of devinfo node",
4299                 devinfo_siblings_walk_init, devinfo_siblings_walk_step, NULL },
4300         { "devi_next", "walk devinfo list",
4301                 NULL, devi_next_walk_step, NULL },
4302         { "devnames", "walk devnames array",
4303                 devnames_walk_init, devnames_walk_step, devnames_walk_fini },
4304         { "minornode", "given a devinfo node, walk minor nodes",
4305                 minornode_walk_init, minornode_walk_step, NULL },
4306         { "softstate",
4307                 "given an i_ddi_soft_state*, list all in-use driver stateps",
4308                 soft_state_walk_init, soft_state_walk_step,
4309                 NULL, NULL },
4310         { "softstate_all",
4311                 "given an i_ddi_soft_state*, list all driver stateps",
4312                 soft_state_walk_init, soft_state_all_walk_step,
4313                 NULL, NULL },
4314         { "devinfo_fmc",
4315                 "walk a fault management handle cache active list",
4316                 devinfo_fmc_walk_init, devinfo_fmc_walk_step, NULL },

4318         /* from group.c */
4319         { "group", "walk all elements of a group",
4320                 group_walk_init, group_walk_step, NULL },

4322         /* from irm.c */
4323         { "irmpools", "walk global list of interrupt pools",
4324                 irmpools_walk_init, list_walk_step, list_walk_fini },
4325         { "irmreqs", "walk list of interrupt requests in an interrupt pool",
4326                 irmreqs_walk_init, list_walk_step, list_walk_fini },

4328         /* from kmem.c */
4329         { "allocdby", "given a thread, walk its allocated bufctls",
4330                 allocdby_walk_init, allocdby_walk_step, allocdby_walk_fini },
4331         { "bufctl", "walk a kmem cache's bufctls",
4332                 bufctl_walk_init, kmem_walk_step, kmem_walk_fini },
4333         { "bufctl_history", "walk the available history of a bufctl",
4334                 bufctl_history_walk_init, bufctl_history_walk_step,
4335                 bufctl_history_walk_fini },
4336         { "freedby", "given a thread, walk its freed bufctls",
4337                 freedby_walk_init, allocdby_walk_step, allocdby_walk_fini },
4338         { "freectl", "walk a kmem cache's free bufctls",
4339                 freectl_walk_init, kmem_walk_step, kmem_walk_fini },
4340         { "freectl_constructed", "walk a kmem cache's constructed free bufctls",
4341                 freectl_constructed_walk_init, kmem_walk_step, kmem_walk_fini },
4342         { "freemem", "walk a kmem cache's free memory",
4343                 freemem_walk_init, kmem_walk_step, kmem_walk_fini },
4344         { "freemem_constructed", "walk a kmem cache's constructed free memory",
4345                 freemem_constructed_walk_init, kmem_walk_step, kmem_walk_fini },
4346         { "kmem", "walk a kmem cache",
4347                 kmem_walk_init, kmem_walk_step, kmem_walk_fini },
4348         { "kmem_cpu_cache", "given a kmem cache, walk its per-CPU caches",
4349                 kmem_cpu_cache_walk_init, kmem_cpu_cache_walk_step, NULL },
4350         { "kmem_hash", "given a kmem cache, walk its allocated hash table",
4351                 kmem_hash_walk_init, kmem_hash_walk_step, kmem_hash_walk_fini },
4352         { "kmem_log", "walk the kmem transaction log",
4353                 kmem_log_walk_init, kmem_log_walk_step, kmem_log_walk_fini },
4354         { "kmem_slab", "given a kmem cache, walk its slabs",
4355                 kmem_slab_walk_init, combined_walk_step, combined_walk_fini },
4356         { "kmem_slab_partial",
4357             "given a kmem cache, walk its partially allocated slabs (min 1)",
4358                 kmem_slab_walk_partial_init, combined_walk_step,
4359                 combined_walk_fini },
4360         { "vmem", "walk vmem structures in pre-fix, depth-first order",
```

```
4361                 vmem_walk_init, vmem_walk_step, vmem_walk_fini },
4362         { "vmem_alloc", "given a vmem_t, walk its allocated vmem_segs",
4363                 vmem_alloc_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },
4364         { "vmem_free", "given a vmem_t, walk its free vmem_segs",
4365                 vmem_free_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },
4366         { "vmem_postfix", "walk vmem structures in post-fix, depth-first order",
4367                 vmem_walk_init, vmem_postfix_walk_step, vmem_walk_fini },
4368         { "vmem_seg", "given a vmem_t, walk all of its vmem_segs",
4369                 vmem_seg_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },
4370         { "vmem_span", "given a vmem_t, walk its spanning vmem_segs",
4371                 vmem_span_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },

4373         /* from ldi.c */
4374         { "ldi_handle", "walk the layered driver handle hash",
4375                 ldi_handle_walk_init, ldi_handle_walk_step, NULL },
4376         { "ldi_ident", "walk the layered driver identifier hash",
4377                 ldi_ident_walk_init, ldi_ident_walk_step, NULL },

4379         /* from leaky.c + leaky_subr.c */
4380         { "leak", "given a leaked bufctl or vmem_seg, find leaks w/ same "
4381             "stack trace",
4382                 leaky_walk_init, leaky_walk_step, leaky_walk_fini },
4383         { "leakbuf", "given a leaked bufctl or vmem_seg, walk buffers for "
4384             "leaks w/ same stack trace",
4385                 leaky_walk_init, leaky_buf_walk_step, leaky_walk_fini },

4387         /* from lgrp.c */
4388         { "lgrp_cpulist", "walk CPUs in a given lgroup",
4389                 lgrp_cpulist_walk_init, lgrp_cpulist_walk_step, NULL },
4390         { "lgrptbl", "walk lgroup table",
4391                 lgrp_walk_init, lgrp_walk_step, NULL },
4392         { "lgrp_parents", "walk up lgroup lineage from given lgroup",
4393                 lgrp_parents_walk_init, lgrp_parents_walk_step, NULL },
4394         { "lgrp_rsrc_mem", "walk lgroup memory resources of given lgroup",
4395                 lgrp_rsrc_mem_walk_init, lgrp_set_walk_step, NULL },
4396         { "lgrp_rsrc_cpu", "walk lgroup CPU resources of given lgroup",
4397                 lgrp_rsrc_cpu_walk_init, lgrp_set_walk_step, NULL },

4399         /* from list.c */
4400         { LIST_WALK_NAME, LIST_WALK_DESC,
4401                 list_walk_init, list_walk_step, list_walk_fini },

4403         /* from mdi.c */
4404         { "mdipi_client_list", "Walker for mdi_pathinfo pi_client_link",
4405                 mdi_pi_client_link_walk_init,
4406                 mdi_pi_client_link_walk_step,
4407                 mdi_pi_client_link_walk_fini },
4408         { "mdipi_phci_list", "Walker for mdi_pathinfo pi_phci_link",
4409                 mdi_pi_phci_link_walk_init,
4410                 mdi_pi_phci_link_walk_step,
4411                 mdi_pi_phci_link_walk_fini },
4412         { "mdiphci_list", "Walker for mdi_phci ph_next link",
4413                 mdi_phci_ph_next_walk_init,
4414                 mdi_phci_ph_next_walk_step,
4415                 mdi_phci_ph_next_walk_fini },

4417         /* from memory.c */
4418         { "allpages", "walk all pages, including free pages",
4419                 allpages_walk_init, allpages_walk_step, allpages_walk_fini },
4420         { "anon", "given an amp, list allocated anon structures",
4421                 anon_walk_init, anon_walk_step, anon_walk_fini,
4422                 ANON_WALK_ALLOC },
4423         { "anon_all", "given an amp, list contents of all anon slots",
4424                 anon_walk_init, anon_walk_step, anon_walk_fini,
4425                 ANON_WALK_ALL },
4426         { "memlist", "walk specified memlist",
```



```
4427                 NULL, memlist_walk_step, NULL },
4428         { "page", "walk all pages, or those from the specified vnode",
4429                 page_walk_init, page_walk_step, page_walk_fini },
4430         { "seg", "given an as, list of segments",
4431                 seg_walk_init, avl_walk_step, avl_walk_fini },
4432         { "segvn_anon",
4433                 "given a struct segvn_data, list allocated anon structures",
4434                 segvn_anon_walk_init, anon_walk_step, anon_walk_fini,
4435                 ANON_WALK_ALLOC },
4436         { "segvn_anon_all",
4437                 "given a struct segvn_data, list contents of all anon slots",
4438                 segvn_anon_walk_init, anon_walk_step, anon_walk_fini,
4439                 ANON_WALK_ALL },
4440         { "segvn_pages",
4441                 "given a struct segvn_data, list resident pages in "
4442                 "offset order",
4443                 segvn_pages_walk_init, segvn_pages_walk_step,
4444                 segvn_pages_walk_fini, SEGVN_PAGES_RESIDENT },
4445         { "segvn_pages_all",
4446                 "for each offset in a struct segvn_data, give page_t pointer "
4447                 "(if resident), or NULL.",
4448                 segvn_pages_walk_init, segvn_pages_walk_step,
4449                 segvn_pages_walk_fini, SEGVN_PAGES_ALL },
4450         { "swapinfo", "walk swapinfo structures",
4451                 swap_walk_init, swap_walk_step, NULL },

4453         /* from mmd.c */
4454         { "pattr", "walk pattr_t structures", pattr_walk_init,
4455                 mmdq_walk_step, mmdq_walk_fini },
4456         { "pdesc", "walk pdesc_t structures",
4457                 pdesc_walk_init, mmdq_walk_step, mmdq_walk_fini },
4458         { "pdesc_slab", "walk pdesc_slab_t structures",
4459                 pdesc_slab_walk_init, mmdq_walk_step, mmdq_walk_fini },

4461         /* from modhash.c */
4462         { "modhash", "walk list of mod_hash structures", modhash_walk_init,
4463                 modhash_walk_step, NULL },
4464         { "modent", "walk list of entries in a given mod_hash",
4465                 modent_walk_init, modent_walk_step, modent_walk_fini },
4466         { "modchain", "walk list of entries in a given mod_hash_entry",
4467                 NULL, modchain_walk_step, NULL },

4469         /* from net.c */
4470         { "icmp", "walk ICMP control structures using MI for all stacks",
4471                 mi_payload_walk_init, mi_payload_walk_step, NULL,
4472                 &mi_icmp_arg },
4473         { "mi", "given a MI_O, walk the MI",
4474                 mi_walk_init, mi_walk_step, mi_walk_fini, NULL },
4475         { "sonode", "given a sonode, walk its children",
4476                 sonode_walk_init, sonode_walk_step, sonode_walk_fini, NULL },
4477         { "icmp_stacks", "walk all the icmp_stack_t",
4478                 icmp_stacks_walk_init, icmp_stacks_walk_step, NULL },
4479         { "tcp_stacks", "walk all the tcp_stack_t",
4480                 tcp_stacks_walk_init, tcp_stacks_walk_step, NULL },
4481         { "udp_stacks", "walk all the udp_stack_t",
4482                 udp_stacks_walk_init, udp_stacks_walk_step, NULL },
4483         { "dccp_stacks", "walk all the dccp_stack_t",
4484                 dccp_stacks_walk_init, dccp_stacks_walk_step, NULL },
4485 #endif /* ! codereview */

4487         /* from netstack.c */
4488         { "netstack", "walk a list of kernel netstacks",
4489                 netstack_walk_init, netstack_walk_step, NULL },

4491         /* from nvpair.c */
4492         { NVPAIR_WALKER_NAME, NVPAIR_WALKER_DESCR,
```

```
4493                 nvpair_walk_init, nvpair_walk_step, NULL },

4495         /* from rctl.c */
4496         { "rctl_dict_list", "walk all rctl_dict_entry_t's from rctl_lists",
4497                 rctl_dict_walk_init, rctl_dict_walk_step, NULL },
4498         { "rctl_set", "given a rctl_set, walk all rctls", rctl_set_walk_init,
4499                 rctl_set_walk_step, NULL },
4500         { "rctl_val", "given a rctl_t, walk all rctl_val entries associated",
4501                 rctl_val_walk_init, rctl_val_walk_step },

4503         /* from sobj.c */
4504         { "blocked", "walk threads blocked on a given sobj",
4505                 blocked_walk_init, blocked_walk_step, NULL },
4506         { "wchan", "given a wchan, list of blocked threads",
4507                 wchan_walk_init, wchan_walk_step, wchan_walk_fini },

4509         /* from stream.c */
4510         { "b_cont", "walk mblk_t list using b_cont",
4511                 mblk_walk_init, b_cont_step, mblk_walk_fini },
4512         { "b_next", "walk mblk_t list using b_next",
4513                 mblk_walk_init, b_next_step, mblk_walk_fini },
4514         { "qlink", "walk queue_t list using q_link",
4515                 queue_walk_init, queue_link_step, queue_walk_fini },
4516         { "qnext", "walk queue_t list using q_next",
4517                 queue_walk_init, queue_next_step, queue_walk_fini },
4518         { "strftblk", "given a dblk_t, walk STREAMS flow trace event list",
4519                 strftblk_walk_init, strftblk_step, strftblk_walk_fini },
4520         { "readq", "walk read queue side of stdata",
4521                 str_walk_init, strr_walk_step, str_walk_fini },
4522         { "writeq", "walk write queue side of stdata",
4523                 str_walk_init, strw_walk_step, str_walk_fini },

4525         /* from taskq.c */
4526         { "taskq_thread", "given a taskq_t, list all of its threads",
4527                 taskq_thread_walk_init,
4528                 taskq_thread_walk_step,
4529                 taskq_thread_walk_fini },
4530         { "taskq_entry", "given a taskq_t*, list all taskq_ent_t in the list",
4531                 taskq_ent_walk_init, taskq_ent_walk_step, NULL },

4533         /* from thread.c */
4534         { "deathrow", "walk threads on both lwp_ and thread_deathrow",
4535                 deathrow_walk_init, deathrow_walk_step, NULL },
4536         { "cpu_dispq", "given a cpu_t, walk threads in dispatcher queues",
4537                 cpu_dispq_walk_init, dispq_walk_step, dispq_walk_fini },
4538         { "cpupart_dispq",
4539                 "given a cpupart_t, walk threads in dispatcher queues",
4540                 cpupart_dispq_walk_init, dispq_walk_step, dispq_walk_fini },
4541         { "lwp_deathrow", "walk lwp_deathrow",
4542                 lwp_deathrow_walk_init, deathrow_walk_step, NULL },
4543         { "thread", "global or per-process kthread_t structures",
4544                 thread_walk_init, thread_walk_step, thread_walk_fini },
4545         { "thread_deathrow", "walk threads on thread_deathrow",
4546                 thread_deathrow_walk_init, deathrow_walk_step, NULL },

4548         /* from tsd.c */
4549         { "tsd", "walk list of thread-specific data",
4550                 tsd_walk_init, tsd_walk_step, tsd_walk_fini },

4552         /* from tsol.c */
4553         { "tnrh", "walk remote host cache structures",
4554             tnrh_walk_init, tnrh_walk_step, tnrh_walk_fini },
4555         { "tnrhtp", "walk remote host template structures",
4556             tnrhtp_walk_init, tnrhtp_walk_step, tnrhtp_walk_fini },

4558         /*
```



```
4559          * typegraph does not work under kmdb, as it requires too much memory
4560          * for its internal data structures.
4561          */
4562 #ifndef _KMDB
4563         /* from typegraph.c */
4564         { "typeconflict", "walk buffers with conflicting type inferences",
4565                 typegraph_walk_init, typeconflict_walk_step },
4566         { "typeunknown", "walk buffers with unknown types",
4567                 typegraph_walk_init, typeunknown_walk_step },
4568 #endif

4570         /* from vfs.c */
4571         { "vfs", "walk file system list",
4572                 vfs_walk_init, vfs_walk_step },

4574         /* from zone.c */
4575         { "zone", "walk a list of kernel zones",
4576                 zone_walk_init, zone_walk_step, NULL },
4577         { "zsd", "walk list of zsd entries for a zone",
4578                 zsd_walk_init, zsd_walk_step, NULL },

4580         { NULL }
4581 };

4583 static const mdb_modinfo_t modinfo = { MDB_API_VERSION, dcmds, walkers };

4585 /*ARGSUSED*/
4586 static void
4587 genunix_statechange_cb(void *ignored)
4588 {
4589         /*
4590          * Force ::findleaks and ::stacks to let go any cached state.
4591          */
4592         leaky_cleanup(1);
4593         stacks_cleanup(1);

4595         kmem_statechange();     /* notify kmem */
4596 }

4598 const mdb_modinfo_t *
4599 _mdb_init(void)
4600 {
4601         kmem_init();

4603         (void) mdb_callback_add(MDB_CALLBACK_STCHG,
4604             genunix_statechange_cb, NULL);

4606         return (&modinfo);
4607 }

4609 void
4610 _mdb_fini(void)
4611 {
4612         leaky_cleanup(1);
4613         stacks_cleanup(1);
4614 }
```

```
**********************************************************
   43068 Mon Jul  9 14:38:09 2012
new/usr/src/cmd/mdb/common/modules/genunix/net.c
dccp: build fixes, mdb (vfs sonode missing)
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <mdb/mdb_modapi.h>
#include <mdb/mdb_ks.h>
#include <mdb/mdb_ctf.h>
#include <sys/types.h>
#include <sys/tihdr.h>
#include <inet/led.h>
#include <inet/common.h>
#include <netinet/in.h>
#include <netinet/ip6.h>
#include <netinet/icmp6.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/ipclassifier.h>
#include <inet/tcp.h>
#include <sys/stream.h>
#include <sys/vfs.h>
#include <sys/stropts.h>
#include <sys/tpicommon.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/cred_impl.h>
#include <inet/udp_impl.h>
#include <inet/rawip_impl.h>
#include <inet/mi.h>
#include <inet/dccp/dccp_impl.h>
#endif /* ! codereview */
#include <fs/sockfs/socktpi_impl.h>
#include <net/bridge_impl.h>
#include <io/trill_impl.h>
#include <sys/mac_impl.h>

#define ADDR_V6_WIDTH   23
#define ADDR_V4_WIDTH   15

#define NETSTAT_ALL     0x01
#define NETSTAT_VERBOSE 0x02
```



```
#define NETSTAT_ROUTE   0x04
#define NETSTAT_V4      0x08
#define NETSTAT_V6      0x10
#define NETSTAT_UNIX    0x20

#define NETSTAT_FIRST   0x80000000u

typedef struct netstat_cb_data_s {
        uint_t  opts;
        conn_t  conn;
        int     af;
} netstat_cb_data_t;

int
icmp_stacks_walk_init(mdb_walk_state_t *wsp)
{
        if (mdb_layered_walk("netstack", wsp) == -1) {
                mdb_warn("can't walk 'netstack'");
                return (WALK_ERR);
        }
        return (WALK_NEXT);
}

int
icmp_stacks_walk_step(mdb_walk_state_t *wsp)
{
        uintptr_t kaddr;
        netstack_t nss;

        if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
                mdb_warn("can't read netstack at %p", wsp->walk_addr);
                return (WALK_ERR);
        }
        kaddr = (uintptr_t)nss.netstack_modules[NS_ICMP];
        return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
}

int
tcp_stacks_walk_init(mdb_walk_state_t *wsp)
{
        if (mdb_layered_walk("netstack", wsp) == -1) {
                mdb_warn("can't walk 'netstack'");
                return (WALK_ERR);
        }
        return (WALK_NEXT);
}

int
tcp_stacks_walk_step(mdb_walk_state_t *wsp)
{
        uintptr_t kaddr;
        netstack_t nss;

        if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
                mdb_warn("can't read netstack at %p", wsp->walk_addr);
                return (WALK_ERR);
        }
        kaddr = (uintptr_t)nss.netstack_modules[NS_TCP];
        return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
}

int
udp_stacks_walk_init(mdb_walk_state_t *wsp)
{
        if (mdb_layered_walk("netstack", wsp) == -1) {
                mdb_warn("can't walk 'netstack'");
```

```
128                 return (WALK_ERR);
129         }
130         return (WALK_NEXT);
131 }

133 int
134 udp_stacks_walk_step(mdb_walk_state_t *wsp)
135 {
136         uintptr_t kaddr;
137         netstack_t nss;

139         if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
140                 mdb_warn("can't read netstack at %p", wsp->walk_addr);
141                 return (WALK_ERR);
142         }
143         kaddr = (uintptr_t)nss.netstack_modules[NS_UDP];
144         return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
145 }

147 int
148 dccp_stacks_walk_init(mdb_walk_state_t *wsp)
149 {
150         if (mdb_layered_walk("netstack", wsp) == -1) {
151                 mdb_warn("can't walk 'netstack'");
152                 return (WALK_ERR);
153         }
154         return (WALK_NEXT);
155 }

157 int
158 dccp_stacks_walk_step(mdb_walk_state_t *wsp)
159 {
160         uintptr_t kaddr;
161         netstack_t nss;

163         if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
164                 mdb_warn("can't read netstack at %p", wsp->walk_addr);
165                 return (WALK_ERR);
166         }
167         kaddr = (uintptr_t)nss.netstack_modules[NS_DCCP];
168         return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
169 }

171 #endif /* ! codereview */
172 /*
173  * Print an IPv4 address and port number in a compact and easy to read format
174  * The arguments are in network byte order
175  */
176 static void
177 net_ipv4addrport_pr(const in6_addr_t *nipv6addr, in_port_t nport)
178 {
179         uint32_t naddr = V4_PART_OF_V6((*nipv6addr));

181         mdb_nhconvert(&nport, &nport, sizeof (nport));
182         mdb_printf("%*I.%-5hu", ADDR_V4_WIDTH, naddr, nport);
183 }

185 /*
186  * Print an IPv6 address and port number in a compact and easy to read format
187  * The arguments are in network byte order
188  */
189 static void
190 net_ipv6addrport_pr(const in6_addr_t *naddr, in_port_t nport)
191 {
192         mdb_nhconvert(&nport, &nport, sizeof (nport));
193         mdb_printf("%*N.%-5hu", ADDR_V6_WIDTH, naddr, nport);
```



```
194 }

196 static int
197 net_tcp_active(const tcp_t *tcp)
198 {
199         return (tcp->tcp_state >= TCPS_ESTABLISHED);
200 }

202 static int
203 net_tcp_ipv4(const tcp_t *tcp)
204 {
205         return ((tcp->tcp_connp->conn_ipversion == IPV4_VERSION) ||
206             (IN6_IS_ADDR_UNSPECIFIED(&tcp->tcp_connp->conn_laddr_v6) &&
207             (tcp->tcp_state <= TCPS_LISTEN)));
208 }

210 static int
211 net_tcp_ipv6(const tcp_t *tcp)
212 {
213         return (tcp->tcp_connp->conn_ipversion == IPV6_VERSION);
214 }

216 static int
217 net_udp_active(const udp_t *udp)
218 {
219         return ((udp->udp_state == TS_IDLE) ||
220             (udp->udp_state == TS_DATA_XFER));
221 }

223 static int
224 net_udp_ipv4(const udp_t *udp)
225 {
226         return ((udp->udp_connp->conn_ipversion == IPV4_VERSION) ||
227             (IN6_IS_ADDR_UNSPECIFIED(&udp->udp_connp->conn_laddr_v6) &&
228             (udp->udp_state <= TS_IDLE)));
229 }

231 static int
232 net_udp_ipv6(const udp_t *udp)
233 {
234         return (udp->udp_connp->conn_ipversion == IPV6_VERSION);
235 }

237 static int
238 net_dccp_active(const dccp_t *dccp)
239 {
240         return ((dccp->dccp_state == TS_IDLE) ||
241             (dccp->dccp_state == TS_DATA_XFER));
242 }

244 static int
245 net_dccp_ipv4(const dccp_t *dccp)
246 {
247         return ((dccp->dccp_connp->conn_ipversion == IPV4_VERSION) ||
248             (IN6_IS_ADDR_UNSPECIFIED(&dccp->dccp_connp->conn_laddr_v6) &&
249             (dccp->dccp_state <= DCCPS_LISTEN)));
250 }

252 static int
253 net_dccp_ipv6(const dccp_t *dccp)
254 {
255         return (dccp->dccp_connp->conn_ipversion == IPV6_VERSION);
256 }

258 #endif /* ! codereview */
259 int
```

```
260 sonode_walk_init(mdb_walk_state_t *wsp)
261 {
262         if (wsp->walk_addr == NULL) {
263                 GElf_Sym sym;
264                 struct socklist *slp;

266                 if (mdb_lookup_by_obj("sockfs", "socklist", &sym) == -1) {
267                         mdb_warn("failed to lookup sockfs`socklist");
268                         return (WALK_ERR);
269                 }

271                 slp = (struct socklist *)(uintptr_t)sym.st_value;

273                 if (mdb_vread(&wsp->walk_addr, sizeof (wsp->walk_addr),
274                     (uintptr_t)&slp->sl_list) == -1) {
275                         mdb_warn("failed to read address of initial sonode "
276                             "at %p", &slp->sl_list);
277                         return (WALK_ERR);
278                 }
279         }

281         wsp->walk_data = mdb_alloc(sizeof (struct sotpi_sonode), UM_SLEEP);
282         return (WALK_NEXT);
283 }

285 int
286 sonode_walk_step(mdb_walk_state_t *wsp)
287 {
288         int status;
289         struct sotpi_sonode *stp;

291         if (wsp->walk_addr == NULL)
292                 return (WALK_DONE);

294         if (mdb_vread(wsp->walk_data, sizeof (struct sotpi_sonode),
295             wsp->walk_addr) == -1) {
296                 mdb_warn("failed to read sonode at %p", wsp->walk_addr);
297                 return (WALK_ERR);
298         }

300         status = wsp->walk_callback(wsp->walk_addr, wsp->walk_data,
301             wsp->walk_cbdata);

303         stp = wsp->walk_data;

305         wsp->walk_addr = (uintptr_t)stp->st_info.sti_next_so;
306         return (status);
307 }

309 void
310 sonode_walk_fini(mdb_walk_state_t *wsp)
311 {
312         mdb_free(wsp->walk_data, sizeof (struct sotpi_sonode));
313 }

315 struct mi_walk_data {
316         uintptr_t mi_wd_miofirst;
317         MI_O mi_wd_miodata;
318 };

320 int
321 mi_walk_init(mdb_walk_state_t *wsp)
322 {
323         struct mi_walk_data *wdp;

325         if (wsp->walk_addr == NULL) {
```



```
326                 mdb_warn("mi doesn't support global walks\n");
327                 return (WALK_ERR);
328         }

330         wdp = mdb_alloc(sizeof (struct mi_walk_data), UM_SLEEP);

332         /* So that we do not immediately return WALK_DONE below */
333         wdp->mi_wd_miofirst = NULL;

335         wsp->walk_data = wdp;
336         return (WALK_NEXT);
337 }

339 int
340 mi_walk_step(mdb_walk_state_t *wsp)
341 {
342         struct mi_walk_data *wdp = wsp->walk_data;
343         MI_OP miop = &wdp->mi_wd_miodata;
344         int status;

346         /* Always false in the first iteration */
347         if ((wsp->walk_addr == (uintptr_t)NULL) ||
348             (wsp->walk_addr == wdp->mi_wd_miofirst)) {
349                 return (WALK_DONE);
350         }

352         if (mdb_vread(miop, sizeof (MI_O), wsp->walk_addr) == -1) {
353                 mdb_warn("failed to read MI object at %p", wsp->walk_addr);
354                 return (WALK_ERR);
355         }

357         /* Only true in the first iteration */
358         if (wdp->mi_wd_miofirst == NULL) {
359                 wdp->mi_wd_miofirst = wsp->walk_addr;
360                 status = WALK_NEXT;
361         } else {
362                 status = wsp->walk_callback(wsp->walk_addr + sizeof (MI_O),
363                     &miop[1], wsp->walk_cbdata);
364         }

366         wsp->walk_addr = (uintptr_t)miop->mi_o_next;
367         return (status);
368 }

370 void
371 mi_walk_fini(mdb_walk_state_t *wsp)
372 {
373         mdb_free(wsp->walk_data, sizeof (struct mi_walk_data));
374 }

376 typedef struct mi_payload_walk_arg_s {
377         const char *mi_pwa_walker;      /* Underlying walker */
378         const off_t mi_pwa_head_off;    /* Offset for mi_o_head_t * in stack */
379         const size_t mi_pwa_size;       /* size of mi payload */
380         const uint_t mi_pwa_flags;      /* device and/or module */
381 } mi_payload_walk_arg_t;

383 #define MI_PAYLOAD_DEVICE       0x1
384 #define MI_PAYLOAD_MODULE       0x2

386 int
387 mi_payload_walk_init(mdb_walk_state_t *wsp)
388 {
389         const mi_payload_walk_arg_t *arg = wsp->walk_arg;

391         if (mdb_layered_walk(arg->mi_pwa_walker, wsp) == -1) {
```

```
392                 mdb_warn("can't walk '%s'", arg->mi_pwa_walker);
393                 return (WALK_ERR);
394         }
395         return (WALK_NEXT);
396 }

398 int
399 mi_payload_walk_step(mdb_walk_state_t *wsp)
400 {
401         const mi_payload_walk_arg_t *arg = wsp->walk_arg;
402         uintptr_t kaddr;

404         kaddr = wsp->walk_addr + arg->mi_pwa_head_off;

406         if (mdb_vread(&kaddr, sizeof (kaddr), kaddr) == -1) {
407                 mdb_warn("can't read address of mi head at %p for %s",
408                     kaddr, arg->mi_pwa_walker);
409                 return (WALK_ERR);
410         }

412         if (kaddr == 0) {
413                 /* Empty list */
414                 return (WALK_DONE);
415         }

417         if (mdb_pwalk("genunix`mi", wsp->walk_callback,
418             wsp->walk_cbdata, kaddr) == -1) {
419                 mdb_warn("failed to walk genunix`mi");
420                 return (WALK_ERR);
421         }
422         return (WALK_NEXT);
423 }

425 const mi_payload_walk_arg_t mi_icmp_arg = {
426         "icmp_stacks", OFFSETOF(icmp_stack_t, is_head), sizeof (icmp_t),
427         MI_PAYLOAD_DEVICE | MI_PAYLOAD_MODULE
428 };

430 int
431 sonode(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
432 {
433         const char *optf = NULL;
434         const char *optt = NULL;
435         const char *optp = NULL;
436         int family, type, proto;
437         int filter = 0;
438         struct sonode so;

440         if (!(flags & DCMD_ADDRSPEC)) {
441                 if (mdb_walk_dcmd("genunix`sonode", "genunix`sonode", argc,
442                     argv) == -1) {
443                         mdb_warn("failed to walk sonode");
444                         return (DCMD_ERR);
445                 }

447                 return (DCMD_OK);
448         }

450         if (mdb_getopts(argc, argv,
451             'f', MDB_OPT_STR, &optf,
452             't', MDB_OPT_STR, &optt,
453             'p', MDB_OPT_STR, &optp,
454             NULL) != argc)
455                 return (DCMD_USAGE);

457         if (optf != NULL) {
```



```
458                 if (strcmp("inet", optf) == 0)
459                         family = AF_INET;
460                 else if (strcmp("inet6", optf) == 0)
461                         family = AF_INET6;
462                 else if (strcmp("unix", optf) == 0)
463                         family = AF_UNIX;
464                 else
465                         family = mdb_strtoull(optf);
466                 filter = 1;
467         }

469         if (optt != NULL) {
470                 if (strcmp("stream", optt) == 0)
471                         type = SOCK_STREAM;
472                 else if (strcmp("dgram", optt) == 0)
473                         type = SOCK_DGRAM;
474                 else if (strcmp("raw", optt) == 0)
475                         type = SOCK_RAW;
476                 else
477                         type = mdb_strtoull(optt);
478                 filter = 1;
479         }

481         if (optp != NULL) {
482                 proto = mdb_strtoull(optp);
483                 filter = 1;
484         }

486         if (DCMD_HDRSPEC(flags) && !filter) {
487                 mdb_printf("%<u>%-?s Family Type Proto State Mode Flag "
488                     "AccessVP%</u>\n", "Sonode:");
489         }

491         if (mdb_vread(&so, sizeof (so), addr) == -1) {
492                 mdb_warn("failed to read sonode at %p", addr);
493                 return (DCMD_ERR);
494         }

496         if ((optf != NULL) && (so.so_family != family))
497                 return (DCMD_OK);

499         if ((optt != NULL) && (so.so_type != type))
500                 return (DCMD_OK);

502         if ((optp != NULL) && (so.so_protocol != proto))
503                 return (DCMD_OK);

505         if (filter) {
506                 mdb_printf("%0?p\n", addr);
507                 return (DCMD_OK);
508         }

510         mdb_printf("%0?p ", addr);

512         switch (so.so_family) {
513         case AF_UNIX:
514                 mdb_printf("unix  ");
515                 break;
516         case AF_INET:
517                 mdb_printf("inet  ");
518                 break;
519         case AF_INET6:
520                 mdb_printf("inet6 ");
521                 break;
522         default:
523                 mdb_printf("%6hi", so.so_family);
```

```
524         }
525
526         switch (so.so_type) {
527         case SOCK_STREAM:
528                 mdb_printf(" strm");
529                 break;
530         case SOCK_DGRAM:
531                 mdb_printf(" dgrm");
532                 break;
533         case SOCK_RAW:
534                 mdb_printf(" raw ");
535                 break;
536         default:
537                 mdb_printf(" %4hi", so.so_type);
538         }
539
540         mdb_printf(" %5hi %05x %04x %04hx\n",
541             so.so_protocol, so.so_state, so.so_mode,
542             so.so_flag);
543
544         return (DCMD_OK);
545 }
546
547 #define MI_PAYLOAD      0x1
548 #define MI_DEVICE       0x2
549 #define MI_MODULE       0x4
550
551 int
552 mi(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
553 {
554         uint_t opts = 0;
555         MI_O    mio;
556
557         if (!(flags & DCMD_ADDRSPEC))
558                 return (DCMD_USAGE);
559
560         if (mdb_getopts(argc, argv,
561             'p', MDB_OPT_SETBITS, MI_PAYLOAD, &opts,
562             'd', MDB_OPT_SETBITS, MI_DEVICE, &opts,
563             'm', MDB_OPT_SETBITS, MI_MODULE, &opts,
564             NULL) != argc)
565                 return (DCMD_USAGE);
566
567         if ((opts & (MI_DEVICE | MI_MODULE)) == (MI_DEVICE | MI_MODULE)) {
568                 mdb_warn("at most one filter, d for devices or m "
569                     "for modules, may be specified\n");
570                 return (DCMD_USAGE);
571         }
572
573         if ((opts == 0) && (DCMD_HDRSPEC(flags))) {
574                 mdb_printf("%<u>%-?s %-?s %-?s IsDev Dev%</u>\n",
575                     "MI_O", "Next", "Prev");
576         }
577
578         if (mdb_vread(&mio, sizeof (mio), addr) == -1) {
579                 mdb_warn("failed to read mi object MI_O at %p", addr);
580                 return (DCMD_ERR);
581         }
582
583         if (opts != 0) {
584                 if (mio.mi_o_isdev == B_FALSE) {
585                         /* mio is a module */
586                         if (!(opts & MI_MODULE) && (opts & MI_DEVICE))
587                                 return (DCMD_OK);
588                 } else {
589                         /* mio is a device */
```



```
590                         if (!(opts & MI_DEVICE) && (opts & MI_MODULE))
591                                 return (DCMD_OK);
592                 }
593
594                 if (opts & MI_PAYLOAD)
595                         mdb_printf("%p\n", addr + sizeof (MI_O));
596                 else
597                         mdb_printf("%p\n", addr);
598                 return (DCMD_OK);
599         }
600
601         mdb_printf("%0?p %0?p %0?p ", addr, mio.mi_o_next, mio.mi_o_prev);
602
603         if (mio.mi_o_isdev == B_FALSE)
604                 mdb_printf("FALSE");
605         else
606                 mdb_printf("TRUE ");
607
608         mdb_printf(" %0?p\n", mio.mi_o_dev);
609
610         return (DCMD_OK);
611 }
612
613 static int
614 ns_to_stackid(uintptr_t kaddr)
615 {
616         netstack_t nss;
617
618         if (mdb_vread(&nss, sizeof (nss), kaddr) == -1) {
619                 mdb_warn("failed to read netstack_t %p", kaddr);
620                 return (0);
621         }
622         return (nss.netstack_stackid);
623 }
624
625
626
627 static void
628 netstat_tcp_verbose_pr(const tcp_t *tcp)
629 {
630         mdb_printf("       %5i %08x %08x %5i %08x %08x %5li %5i\n",
631             tcp->tcp_swnd, tcp->tcp_snxt, tcp->tcp_suna, tcp->tcp_rwnd,
632             tcp->tcp_rack, tcp->tcp_rnxt, tcp->tcp_rto, tcp->tcp_mss);
633 }
634
635 /*ARGSUSED*/
636 static int
637 netstat_tcp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
638 {
639         netstat_cb_data_t *ncb = cb_data;
640         uint_t opts = ncb->opts;
641         int af = ncb->af;
642         uintptr_t tcp_kaddr;
643         conn_t *connp = &ncb->conn;
644         tcp_t tcps, *tcp;
645
646         if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
647                 mdb_warn("failed to read conn_t at %p", kaddr);
648                 return (WALK_ERR);
649         }
650
651         tcp_kaddr = (uintptr_t)connp->conn_tcp;
652         if (mdb_vread(&tcps, sizeof (tcp_t), tcp_kaddr) == -1) {
653                 mdb_warn("failed to read tcp_t at %p", tcp_kaddr);
654                 return (WALK_ERR);
655         }
```

```
657         tcp = &tcps;
658         connp->conn_tcp = tcp;
659         tcp->tcp_connp = connp;
660
661         if (!((opts & NETSTAT_ALL) || net_tcp_active(tcp)) ||
662             (af == AF_INET && !net_tcp_ipv4(tcp)) ||
663             (af == AF_INET6 && !net_tcp_ipv6(tcp))) {
664                 return (WALK_NEXT);
665         }
666
667         mdb_printf("%0?p %2i ", tcp_kaddr, tcp->tcp_state);
668         if (af == AF_INET) {
669                 net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
670                 mdb_printf(" ");
671                 net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
672         } else if (af == AF_INET6) {
673                 net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
674                 mdb_printf(" ");
675                 net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
676         }
677         mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
678         mdb_printf(" %4i\n", connp->conn_zoneid);
679         if (opts & NETSTAT_VERBOSE)
680                 netstat_tcp_verbose_pr(tcp);
681
682         return (WALK_NEXT);
683 }
684
685 /*ARGSUSED*/
686 static int
687 netstat_udp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
688 {
689         netstat_cb_data_t *ncb = cb_data;
690         uint_t opts = ncb->opts;
691         int af = ncb->af;
692         udp_t udp;
693         conn_t *connp = &ncb->conn;
694         char *state;
695
696         if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
697                 mdb_warn("failed to read conn_t at %p", kaddr);
698                 return (WALK_ERR);
699         }
700
701         if (mdb_vread(&udp, sizeof (udp_t),
702             (uintptr_t)connp->conn_udp) == -1) {
703                 mdb_warn("failed to read conn_udp at %p",
704                     (uintptr_t)connp->conn_udp);
705                 return (WALK_ERR);
706         }
707
708         connp->conn_udp = &udp;
709         udp.udp_connp = connp;
710
711         if (!((opts & NETSTAT_ALL) || net_udp_active(&udp)) ||
712             (af == AF_INET && !net_udp_ipv4(&udp)) ||
713             (af == AF_INET6 && !net_udp_ipv6(&udp))) {
714                 return (WALK_NEXT);
715         }
716
717         if (udp.udp_state == TS_UNBND)
718                 state = "UNBOUND";
719         else if (udp.udp_state == TS_IDLE)
720                 state = "IDLE";
721         else if (udp.udp_state == TS_DATA_XFER)
```



```
722                 state = "CONNECTED";
723         else
724                 state = "UNKNOWN";
725
726         mdb_printf("%0?p %10s ", (uintptr_t)connp->conn_udp, state);
727         if (af == AF_INET) {
728                 net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
729                 mdb_printf(" ");
730                 net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
731         } else if (af == AF_INET6) {
732                 net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
733                 mdb_printf(" ");
734                 net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
735         }
736         mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
737         mdb_printf(" %4i\n", connp->conn_zoneid);
738
739         return (WALK_NEXT);
740 }
741
742 /*ARGSUSED*/
743 static int
744 netstat_icmp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
745 {
746         netstat_cb_data_t *ncb = cb_data;
747         int af = ncb->af;
748         icmp_t icmp;
749         conn_t *connp = &ncb->conn;
750         char *state;
751
752         if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
753                 mdb_warn("failed to read conn_t at %p", kaddr);
754                 return (WALK_ERR);
755         }
756
757         if (mdb_vread(&icmp, sizeof (icmp_t),
758             (uintptr_t)connp->conn_icmp) == -1) {
759                 mdb_warn("failed to read conn_icmp at %p",
760                     (uintptr_t)connp->conn_icmp);
761                 return (WALK_ERR);
762         }
763
764         connp->conn_icmp = &icmp;
765         icmp.icmp_connp = connp;
766
767         if ((af == AF_INET && connp->conn_ipversion != IPV4_VERSION) ||
768             (af == AF_INET6 && connp->conn_ipversion != IPV6_VERSION)) {
769                 return (WALK_NEXT);
770         }
771
772         if (icmp.icmp_state == TS_UNBND)
773                 state = "UNBOUND";
774         else if (icmp.icmp_state == TS_IDLE)
775                 state = "IDLE";
776         else if (icmp.icmp_state == TS_DATA_XFER)
777                 state = "CONNECTED";
778         else
779                 state = "UNKNOWN";
780
781         mdb_printf("%0?p %10s ", (uintptr_t)connp->conn_icmp, state);
782         if (af == AF_INET) {
783                 net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
784                 mdb_printf(" ");
785                 net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
786         } else if (af == AF_INET6) {
787                 net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
```

```
788                 mdb_printf(" ");
789                 net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
790         }
791         mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
792         mdb_printf(" %4i\n", connp->conn_zoneid);

794         return (WALK_NEXT);
795 }

797 static void
798 netstat_dccp_verbose_pr(const dccp_t *dccp)
799 {
800 /* XXX:DCCP
801         mdb_printf("       %5i %08x %08x %5i %08x %08x %5li %5i\n",
802             tcp->tcp_swnd, tcp->tcp_snxt, tcp->tcp_suna, tcp->tcp_rwnd,
803             tcp->tcp_rack, tcp->tcp_rnxt, tcp->tcp_rto, tcp->tcp_mss);
804 */
805 }

807 /*ARGSUSED*/
808 static int
809 netstat_dccp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
810 {
811         netstat_cb_data_t *ncb = cb_data;
812         uint_t opts = ncb->opts;
813         int af = ncb->af;
814         uintptr_t dccp_kaddr;
815         conn_t *connp = &ncb->conn;
816         dccp_t dccps, *dccp;

818         if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
819                 mdb_warn("failed to read conn_t at %p", kaddr);
820                 return (WALK_ERR);
821         }

823         dccp_kaddr = (uintptr_t)connp->conn_dccp;
824         if (mdb_vread(&dccps, sizeof (dccp_t), dccp_kaddr) == -1) {
825                 mdb_warn("failed to read tcp_t at %p", dccp_kaddr);
826                 return (WALK_ERR);
827         }

829         dccp = &dccps;
830         connp->conn_dccp = dccp;
831         dccp->dccp_connp = connp;

833         if (!((opts & NETSTAT_ALL) || net_dccp_active(dccp)) ||
834             (af == AF_INET && !net_dccp_ipv4(dccp)) ||
835             (af == AF_INET6 && !net_dccp_ipv6(dccp))) {
836                 return (WALK_NEXT);
837         }

839         mdb_printf("%0?p %2i ", dccp_kaddr, dccp->dccp_state);
840         if (af == AF_INET) {
841                 net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
842                 mdb_printf(" ");
843                 net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
844         } else if (af == AF_INET6) {
845                 net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
846                 mdb_printf(" ");
847                 net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
848         }
849         mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
850         mdb_printf(" %4i\n", connp->conn_zoneid);
851         if (opts & NETSTAT_VERBOSE)
852                 netstat_dccp_verbose_pr(dccp);
```



```
854         return (WALK_NEXT);
855 }

857 #endif /* ! codereview */
858 /*
859  * print the address of a unix domain socket
860  *
861  * so is the address of a AF_UNIX struct sonode in mdb's address space
862  * soa is the address of the struct soaddr to print
863  *
864  * returns 0 on success, -1 otherwise
865  */
866 static int
867 netstat_unix_name_pr(const struct sotpi_sonode *st, const struct soaddr *soa)
868 {
869         const struct sonode *so = &st->st_sonode;
870         const char none[] = " (none)";

872         if ((so->so_state & SS_ISBOUND) && (soa->soa_len != 0)) {
873                 if (st->st_info.sti_faddr_noxlate) {
874                         mdb_printf("%-14s ", " (socketpair)");
875                 } else {
876                         if (soa->soa_len > sizeof (sa_family_t)) {
877                                 char addr[MAXPATHLEN + 1];

879                                 if (mdb_readstr(addr, sizeof (addr),
880                                     (uintptr_t)&soa->soa_sa->sa_data) == -1) {
881                                         mdb_warn("failed to read unix address "
882                                             "at %p", &soa->soa_sa->sa_data);
883                                         return (-1);
884                                 }

886                                 mdb_printf("%-14s ", addr);
887                         } else {
888                                 mdb_printf("%-14s ", none);
889                         }
890                 }
891         } else {
892                 mdb_printf("%-14s ", none);
893         }

895         return (0);
896 }

898 /* based on sockfs_snapshot */
899 /*ARGSUSED*/
900 static int
901 netstat_unix_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
902 {
903         const struct sotpi_sonode *st = walk_data;
904         const struct sonode *so = &st->st_sonode;
905         const struct sotpi_info *sti = &st->st_info;

907         if (so->so_count == 0)
908                 return (WALK_NEXT);

910         if (so->so_family != AF_UNIX) {
911                 mdb_warn("sonode of family %hi at %p\n", so->so_family, kaddr);
912                 return (WALK_ERR);
913         }

915         mdb_printf("%-?p ", kaddr);

917         switch (sti->sti_serv_type) {
918         case T_CLTS:
919                 mdb_printf("%-10s ", "dgram");
```

```
920                 break;
921         case T_COTS:
922                 mdb_printf("%-10s ", "stream");
923                 break;
924         case T_COTS_ORD:
925                 mdb_printf("%-10s ", "stream-ord");
926                 break;
927         default:
928                 mdb_printf("%-10i ", sti->sti_serv_type);
929         }

931         if ((so->so_state & SS_ISBOUND) &&
932             (sti->sti_ux_laddr.soua_magic == SOU_MAGIC_EXPLICIT)) {
933                 mdb_printf("%0?p ", sti->sti_ux_laddr.soua_vp);
934         } else {
935                 mdb_printf("%0?p ", NULL);
936         }

938         if ((so->so_state & SS_ISCONNECTED) &&
939             (sti->sti_ux_faddr.soua_magic == SOU_MAGIC_EXPLICIT)) {
940                 mdb_printf("%0?p ", sti->sti_ux_faddr.soua_vp);
941         } else {
942                 mdb_printf("%0?p ", NULL);
943         }

945         if (netstat_unix_name_pr(st, &sti->sti_laddr) == -1)
946                 return (WALK_ERR);

948         if (netstat_unix_name_pr(st, &sti->sti_faddr) == -1)
949                 return (WALK_ERR);

951         mdb_printf("%4i\n", so->so_zoneid);

953         return (WALK_NEXT);
954 }

956 static void
957 netstat_tcp_verbose_header_pr(void)
958 {
959         mdb_printf("       %<u>%-5s %-8s %-8s %-5s %-8s %-8s %5s %5s%%</u>\n",
960             "Swind", "Snext", "Suna", "Rwind", "Rack", "Rnext", "Rto", "Mss");
961 }

963 static void
964 get_ifname(const ire_t *ire, char *intf)
965 {
966         ill_t ill;

968         *intf = '\0';
969         if (ire->ire_ill != NULL) {
970                 if (mdb_vread(&ill, sizeof (ill),
971                     (uintptr_t)ire->ire_ill) == -1)
972                         return;
973                 (void) mdb_readstr(intf, MIN(LIFNAMSIZ, ill.ill_name_length),
974                     (uintptr_t)ill.ill_name);
975         }
976 }

978 const in6_addr_t ipv6_all_ones =
979         { 0xffffffffU, 0xffffffffU, 0xffffffffU, 0xffffffffU };

981 static void
982 get_ireflags(const ire_t *ire, char *flags)
983 {
984         (void) strcpy(flags, "U");
985         /* RTF_INDIRECT wins over RTF_GATEWAY - don't display both */
```



```
986         if (ire->ire_flags & RTF_INDIRECT)
987                 (void) strcat(flags, "I");
988         else if (ire->ire_type & IRE_OFFLINK)
989                 (void) strcat(flags, "G");

991         /* IRE_IF_CLONE wins over RTF_HOST - don't display both */
992         if (ire->ire_type & IRE_IF_CLONE)
993                 (void) strcat(flags, "C");
994         else if (ire->ire_ipversion == IPV4_VERSION) {
995                 if (ire->ire_mask == IP_HOST_MASK)
996                         (void) strcat(flags, "H");
997         } else {
998                 if (IN6_ARE_ADDR_EQUAL(&ire->ire_mask_v6, &ipv6_all_ones))
999                         (void) strcat(flags, "H");
1000        }

1002        if (ire->ire_flags & RTF_DYNAMIC)
1003                (void) strcat(flags, "D");
1004        if (ire->ire_type == IRE_BROADCAST)
1005                (void) strcat(flags, "b");
1006        if (ire->ire_type == IRE_MULTICAST)
1007                (void) strcat(flags, "m");
1008        if (ire->ire_type == IRE_LOCAL)
1009                (void) strcat(flags, "L");
1010        if (ire->ire_type == IRE_NOROUTE)
1011                (void) strcat(flags, "N");
1012        if (ire->ire_flags & RTF_MULTIRT)
1013                (void) strcat(flags, "M");
1014        if (ire->ire_flags & RTF_SETSRC)
1015                (void) strcat(flags, "S");
1016        if (ire->ire_flags & RTF_REJECT)
1017                (void) strcat(flags, "R");
1018        if (ire->ire_flags & RTF_BLACKHOLE)
1019                (void) strcat(flags, "B");
1020 }

1022 static int
1023 netstat_irev4_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
1024 {
1025         const ire_t *ire = walk_data;
1026         uint_t *opts = cb_data;
1027         ipaddr_t gate;
1028         char flags[10], intf[LIFNAMSIZ + 1];

1030         if (ire->ire_ipversion != IPV4_VERSION)
1031                 return (WALK_NEXT);

1033         /* Skip certain IREs by default */
1034         if (!(*opts & NETSTAT_ALL) &&
1035             (ire->ire_type &
1036             (IRE_BROADCAST|IRE_LOCAL|IRE_MULTICAST|IRE_NOROUTE|IRE_IF_CLONE)))
1037                 return (WALK_NEXT);

1039         if (*opts & NETSTAT_FIRST) {
1040                 *opts &= ~NETSTAT_FIRST;
1041                 mdb_printf("%<u>%s Table: IPv4%</u>\n",
1042                     (*opts & NETSTAT_VERBOSE) ? "IRE" : "Routing");
1043                 if (*opts & NETSTAT_VERBOSE) {
1044                         mdb_printf("%<u>%-?s %-*s %-*s %-*s Device Mxfrg Rtt  "
1045                             " Ref Flg Out   In/Fwd%</u>\n",
1046                             "Address", ADDR_V4_WIDTH, "Destination",
1047                             ADDR_V4_WIDTH, "Mask", ADDR_V4_WIDTH, "Gateway");
1048                 } else {
1049                         mdb_printf("%<u>%-?s %-*s %-*s Flags Ref  Use   "
1050                             "Interface%</u>\n",
1051                             "Address", ADDR_V4_WIDTH, "Destination",
```

```
1052                            ADDR_V4_WIDTH, "Gateway");
1053                }
1054        }

1056        gate = ire->ire_gateway_addr;

1058        get_ireflags(ire, flags);

1060        get_ifname(ire, intf);

1062        if (*opts & NETSTAT_VERBOSE) {
1063                mdb_printf("%?p %-*I %-*I %-*I %-6s %5u%c %4u %3u %-3s %5u "
1064                    "%u\n", kaddr, ADDR_V4_WIDTH, ire->ire_addr, ADDR_V4_WIDTH,
1065                    ire->ire_mask, ADDR_V4_WIDTH, gate, intf,
1066                    0, ' ',
1067                    ire->ire_metrics.iulp_rtt, ire->ire_refcnt, flags,
1068                    ire->ire_ob_pkt_count, ire->ire_ib_pkt_count);
1069        } else {
1070                mdb_printf("%?p %-*I %-*I %-5s %4u %5u %s\n", kaddr,
1071                    ADDR_V4_WIDTH, ire->ire_addr, ADDR_V4_WIDTH, gate, flags,
1072                    ire->ire_refcnt,
1073                    ire->ire_ob_pkt_count + ire->ire_ib_pkt_count, intf);
1074        }

1076        return (WALK_NEXT);
1077 }

1079 int
1080 ip_mask_to_plen_v6(const in6_addr_t *v6mask)
1081 {
1082        int plen;
1083        int i;
1084        uint32_t val;

1086        for (i = 3; i >= 0; i--)
1087                if (v6mask->s6_addr32[i] != 0)
1088                        break;
1089        if (i < 0)
1090                return (0);
1091        plen = 32 + 32 * i;
1092        val = v6mask->s6_addr32[i];
1093        while (!(val & 1)) {
1094                val >>= 1;
1095                plen--;
1096        }

1098        return (plen);
1099 }

1101 static int
1102 netstat_irev6_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
1103 {
1104        const ire_t *ire = walk_data;
1105        uint_t *opts = cb_data;
1106        const in6_addr_t *gatep;
1107        char deststr[ADDR_V6_WIDTH + 5];
1108        char flags[10], intf[LIFNAMSIZ + 1];
1109        int masklen;

1111        if (ire->ire_ipversion != IPV6_VERSION)
1112                return (WALK_NEXT);

1114        /* Skip certain IREs by default */
1115        if (!(*opts & NETSTAT_ALL) &&
1116            (ire->ire_type &
1117            (IRE_BROADCAST|IRE_LOCAL|IRE_MULTICAST|IRE_NOROUTE|IRE_IF_CLONE)))
```



```
1118                return (WALK_NEXT);

1120        if (*opts & NETSTAT_FIRST) {
1121                *opts &= ~NETSTAT_FIRST;
1122                mdb_printf("\n%<u>%s Table: IPv6%</u>\n",
1123                    (*opts & NETSTAT_VERBOSE) ? "IRE" : "Routing");
1124                if (*opts & NETSTAT_VERBOSE) {
1125                        mdb_printf("%<u>%-?s %-*s %-*s If    PMTU   Rtt   Ref "
1126                            "Flags Out    In/Fwd%</u>\n",
1127                            "Address", ADDR_V6_WIDTH+4, "Destination/Mask",
1128                            ADDR_V6_WIDTH, "Gateway");
1129                } else {
1130                        mdb_printf("%<u>%-?s %-*s %-*s Flags Ref Use    If"
1131                            "%</u>\n",
1132                            "Address", ADDR_V6_WIDTH+4, "Destination/Mask",
1133                            ADDR_V6_WIDTH, "Gateway");
1134                }
1135        }

1137        gatep = &ire->ire_gateway_addr_v6;

1139        masklen = ip_mask_to_plen_v6(&ire->ire_mask_v6);
1140        (void) mdb_snprintf(deststr, sizeof (deststr), "%N/%d",
1141            &ire->ire_addr_v6, masklen);

1143        get_ireflags(ire, flags);

1145        get_ifname(ire, intf);

1147        if (*opts & NETSTAT_VERBOSE) {
1148                mdb_printf("%?p %-*s %-*N %-5s %5u%c %5u %3u %-5s %6u %u\n",
1149                    kaddr, ADDR_V6_WIDTH+4, deststr, ADDR_V6_WIDTH, gatep,
1150                    intf, 0, ' ',
1151                    ire->ire_metrics.iulp_rtt, ire->ire_refcnt,
1152                    flags, ire->ire_ob_pkt_count, ire->ire_ib_pkt_count);
1153        } else {
1154                mdb_printf("%?p %-*s %-*N %-5s %3u %6u %s\n", kaddr,
1155                    ADDR_V6_WIDTH+4, deststr, ADDR_V6_WIDTH, gatep, flags,
1156                    ire->ire_refcnt,
1157                    ire->ire_ob_pkt_count + ire->ire_ib_pkt_count, intf);
1158        }

1160        return (WALK_NEXT);
1161 }

1163 static void
1164 netstat_header_v4(int proto)
1165 {
1166        if (proto == IPPROTO_TCP)
1167                mdb_printf("%<u>%-?s ", "TCPv4");
1168        else if (proto == IPPROTO_UDP)
1169                mdb_printf("%<u>%-?s ", "UDPv4");
1170        else if (proto == IPPROTO_ICMP)
1171                mdb_printf("%<u>%-?s ", "ICMPv4");
1172        mdb_printf("State %6s%*s %6s%*s %-5s %-4s%</u>\n",
1173            "", ADDR_V4_WIDTH, "Local Address",
1174            "", ADDR_V4_WIDTH, "Remote Address", "Stack", "Zone");
1175 }

1177 static void
1178 netstat_header_v6(int proto)
1179 {
1180        if (proto == IPPROTO_TCP)
1181                mdb_printf("%<u>%-?s ", "TCPv6");
1182        else if (proto == IPPROTO_UDP)
1183                mdb_printf("%<u>%-?s ", "UDPv6");
```

```
1184         else if (proto == IPPROTO_ICMP)
1185                 mdb_printf("%<u>%-?s ", "ICMPv6");
1186         mdb_printf("State %6s%*s %6s%*s %-5s %-4s%</u>\n",
1187             "", ADDR_V6_WIDTH, "Local Address",
1188             "", ADDR_V6_WIDTH, "Remote Address", "Stack", "Zone");
1189 }

1191 static int
1192 netstat_print_conn(const char *cache, int proto, mdb_walk_cb_t cbfunc,
1193     void *cbdata)
1194 {
1195         netstat_cb_data_t *ncb = cbdata;

1197         if ((ncb->opts & NETSTAT_VERBOSE) && proto == IPPROTO_TCP)
1198                 netstat_tcp_verbose_header_pr();
1199         if (mdb_walk(cache, cbfunc, cbdata) == -1) {
1200                 mdb_warn("failed to walk %s", cache);
1201                 return (DCMD_ERR);
1202         }
1203         return (DCMD_OK);
1204 }

1206 static int
1207 netstat_print_common(const char *cache, int proto, mdb_walk_cb_t cbfunc,
1208     void *cbdata)
1209 {
1210         netstat_cb_data_t *ncb = cbdata;
1211         int af = ncb->af;
1212         int status = DCMD_OK;

1214         if (af != AF_INET6) {
1215                 ncb->af = AF_INET;
1216                 netstat_header_v4(proto);
1217                 status = netstat_print_conn(cache, proto, cbfunc, cbdata);
1218         }
1219         if (status == DCMD_OK && af != AF_INET) {
1220                 ncb->af = AF_INET6;
1221                 netstat_header_v6(proto);
1222                 status = netstat_print_conn(cache, proto, cbfunc, cbdata);
1223         }
1224         ncb->af = af;
1225         return (status);
1226 }

1228 /*ARGSUSED*/
1229 int
1230 netstat(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1231 {
1232         uint_t opts = 0;
1233         const char *optf = NULL;
1234         const char *optP = NULL;
1235         netstat_cb_data_t *cbdata;
1236         int status;
1237         int af = 0;

1239         if (mdb_getopts(argc, argv,
1240             'a', MDB_OPT_SETBITS, NETSTAT_ALL, &opts,
1241             'f', MDB_OPT_STR, &optf,
1242             'P', MDB_OPT_STR, &optP,
1243             'r', MDB_OPT_SETBITS, NETSTAT_ROUTE, &opts,
1244             'v', MDB_OPT_SETBITS, NETSTAT_VERBOSE, &opts,
1245             NULL) != argc)
1246                 return (DCMD_USAGE);

1248         if (optP != NULL) {
1249                 if ((strcmp("tcp", optP) != 0) && (strcmp("udp", optP) != 0) &&
```



```
1250                     (strcmp("icmp", optP) != 0))
1251                         return (DCMD_USAGE);
1252                 if (opts & NETSTAT_ROUTE)
1253                         return (DCMD_USAGE);
1254         }

1256         if (optf == NULL)
1257                 opts |= NETSTAT_V4 | NETSTAT_V6 | NETSTAT_UNIX;
1258         else if (strcmp("inet", optf) == 0)
1259                 opts |= NETSTAT_V4;
1260         else if (strcmp("inet6", optf) == 0)
1261                 opts |= NETSTAT_V6;
1262         else if (strcmp("unix", optf) == 0)
1263                 opts |= NETSTAT_UNIX;
1264         else
1265                 return (DCMD_USAGE);

1267         if (opts & NETSTAT_ROUTE) {
1268                 if (!(opts & (NETSTAT_V4|NETSTAT_V6)))
1269                         return (DCMD_USAGE);
1270                 if (opts & NETSTAT_V4) {
1271                         opts |= NETSTAT_FIRST;
1272                         if (mdb_walk("ip`ire", netstat_irev4_cb, &opts) == -1) {
1273                                 mdb_warn("failed to walk ip`ire");
1274                                 return (DCMD_ERR);
1275                         }
1276                 }
1277                 if (opts & NETSTAT_V6) {
1278                         opts |= NETSTAT_FIRST;
1279                         if (mdb_walk("ip`ire", netstat_irev6_cb, &opts) == -1) {
1280                                 mdb_warn("failed to walk ip`ire");
1281                                 return (DCMD_ERR);
1282                         }
1283                 }
1284                 return (DCMD_OK);
1285         }

1287         if ((opts & NETSTAT_UNIX) && (optP == NULL)) {
1288                 /* Print Unix Domain Sockets */
1289                 mdb_printf("%<u>%-?s %-10s %-?s %-?s %-14s %-14s %s%</u>\n",
1290                     "AF_UNIX", "Type", "Vnode", "Conn", "Local Addr",
1291                     "Remote Addr", "Zone");

1293                 if (mdb_walk("genunix`sonode", netstat_unix_cb, NULL) == -1) {
1294                         mdb_warn("failed to walk genunix`sonode");
1295                         return (DCMD_ERR);
1296                 }
1297                 if (!(opts & (NETSTAT_V4 | NETSTAT_V6)))
1298                         return (DCMD_OK);
1299         }

1301         cbdata = mdb_alloc(sizeof (netstat_cb_data_t), UM_SLEEP);
1302         cbdata->opts = opts;
1303         if ((optf != NULL) && (opts & NETSTAT_V4))
1304                 af = AF_INET;
1305         else if ((optf != NULL) && (opts & NETSTAT_V6))
1306                 af = AF_INET6;

1308         cbdata->af = af;
1309         if ((optP == NULL) || (strcmp("tcp", optP) == 0)) {
1310                 status = netstat_print_common("tcp_conn_cache", IPPROTO_TCP,
1311                     netstat_tcp_cb, cbdata);
1312                 if (status != DCMD_OK)
1313                         goto out;
1314         }
```

```
1316         if ((optP == NULL) || (strcmp("udp", optP) == 0)) {
1317                 status = netstat_print_common("udp_conn_cache", IPPROTO_UDP,
1318                     netstat_udp_cb, cbdata);
1319                 if (status != DCMD_OK)
1320                         goto out;
1321         }

1323         if ((optP == NULL) || (strcmp("icmp", optP) == 0)) {
1324                 status = netstat_print_common("rawip_conn_cache", IPPROTO_ICMP,
1325                     netstat_icmp_cb, cbdata);
1326                 if (status != DCMD_OK)
1327                         goto out;
1328         }

1330         if ((optP == NULL) || (strcmp("dccp", optP) == 0)) {
1331                 status = netstat_print_common("dccp_conn_cache", IPPROTO_DCCP,
1332                     netstat_dccp_cb, cbdata);
1333                 if (status != DCMD_OK)
1334                         goto out;
1335         }
1336 #endif /* ! codereview */
1337 out:
1338         mdb_free(cbdata, sizeof (netstat_cb_data_t));
1339         return (status);
1340 }

1342 /*
1343  * "::dladm show-bridge" support
1344  */
1345 typedef struct {
1346         uint_t opt_l;
1347         uint_t opt_f;
1348         uint_t opt_t;
1349         const char *name;
1350         clock_t lbolt;
1351         boolean_t found;
1352         uint_t nlinks;
1353         uint_t nfwd;

1355         /*
1356          * These structures are kept inside the 'args' for allocation reasons.
1357          * They're all large data structures (over 1K), and may cause the stack
1358          * to explode.  mdb and kmdb will fail in these cases, and thus we
1359          * allocate them from the heap.
1360          */
1361         trill_inst_t ti;
1362         bridge_link_t bl;
1363         mac_impl_t mi;
1364 } show_bridge_args_t;

1366 static void
1367 show_vlans(const uint8_t *vlans)
1368 {
1369         int i, bit;
1370         uint8_t val;
1371         int rstart = -1, rnext = -1;

1373         for (i = 0; i < BRIDGE_VLAN_ARR_SIZE; i++) {
1374                 val = vlans[i];
1375                 if (i == 0)
1376                         val &= ~1;
1377                 while ((bit = mdb_ffs(val)) != 0) {
1378                         bit--;
1379                         val &= ~(1 << bit);
1380                         bit += i * sizeof (*vlans) * NBBY;
1381                         if (bit != rnext) {
```



```
1382                                 if (rnext != -1 && rstart + 1 != rnext)
1383                                         mdb_printf("-%d", rnext - 1);
1384                                 if (rstart != -1)
1385                                         mdb_printf(",");
1386                                 mdb_printf("%d", bit);
1387                                 rstart = bit;
1388                         }
1389                         rnext = bit + 1;
1390                 }
1391         }
1392         if (rnext != -1 && rstart + 1 != rnext)
1393                 mdb_printf("-%d", rnext - 1);
1394         mdb_printf("\n");
1395 }

1397 /*
1398  * This callback is invoked by a walk of the links attached to a bridge.  If
1399  * we're showing link details, then they're printed here.  If not, then we just
1400  * count up the links for the bridge summary.
1401  */
1402 static int
1403 do_bridge_links(uintptr_t addr, const void *data, void *ptr)
1404 {
1405         show_bridge_args_t *args = ptr;
1406         const bridge_link_t *blp = data;
1407         char macaddr[ETHERADDRL * 3];
1408         const char *name;

1410         args->nlinks++;

1412         if (!args->opt_l)
1413                 return (WALK_NEXT);

1415         if (mdb_vread(&args->mi, sizeof (args->mi),
1416             (uintptr_t)blp->bl_mh) == -1) {
1417                 mdb_warn("cannot read mac data at %p", blp->bl_mh);
1418                 name = "?";
1419         } else  {
1420                 name = args->mi.mi_name;
1421         }

1423         mdb_mac_addr(blp->bl_local_mac, ETHERADDRL, macaddr,
1424             sizeof (macaddr));

1426         mdb_printf("%-?p %-16s %-17s %03X %-4d ", addr, name, macaddr,
1427             blp->bl_flags, blp->bl_pvid);

1429         if (blp->bl_trilldata == NULL) {
1430                 switch (blp->bl_state) {
1431                 case BLS_BLOCKLISTEN:
1432                         name = "BLOCK";
1433                         break;
1434                 case BLS_LEARNING:
1435                         name = "LEARN";
1436                         break;
1437                 case BLS_FORWARDING:
1438                         name = "FWD";
1439                         break;
1440                 default:
1441                         name = "?";
1442                 }
1443                 mdb_printf("%-5s ", name);
1444                 show_vlans(blp->bl_vlans);
1445         } else {
1446                 show_vlans(blp->bl_afs);
1447         }
```

```
1449         return (WALK_NEXT);
1450 }

1452 /*
1453  * It seems a shame to duplicate this code, but merging it with the link
1454  * printing code above is more trouble than it would be worth.
1455  */
1456 static void
1457 print_link_name(show_bridge_args_t *args, uintptr_t addr, char sep)
1458 {
1459         const char *name;

1461         if (mdb_vread(&args->bl, sizeof (args->bl), addr) == -1) {
1462                 mdb_warn("cannot read bridge link at %p", addr);
1463                 return;
1464         }

1466         if (mdb_vread(&args->mi, sizeof (args->mi),
1467             (uintptr_t)args->bl.bl_mh) == -1) {
1468                 name = "?";
1469         } else  {
1470                 name = args->mi.mi_name;
1471         }

1473         mdb_printf("%s%c", name, sep);
1474 }

1476 static int
1477 do_bridge_fwd(uintptr_t addr, const void *data, void *ptr)
1478 {
1479         show_bridge_args_t *args = ptr;
1480         const bridge_fwd_t *bfp = data;
1481         char macaddr[ETHERADDRL * 3];
1482         int i;
1483 #define MAX_FWD_LINKS   16
1484         bridge_link_t *links[MAX_FWD_LINKS];
1485         uint_t nlinks;

1487         args->nfwd++;

1489         if (!args->opt_f)
1490                 return (WALK_NEXT);

1492         if ((nlinks = bfp->bf_nlinks) > MAX_FWD_LINKS)
1493                 nlinks = MAX_FWD_LINKS;

1495         if (mdb_vread(links, sizeof (links[0]) * nlinks,
1496             (uintptr_t)bfp->bf_links) == -1) {
1497                 mdb_warn("cannot read bridge forwarding links at %p",
1498                     bfp->bf_links);
1499                 return (WALK_ERR);
1500         }

1502         mdb_mac_addr(bfp->bf_dest, ETHERADDRL, macaddr, sizeof (macaddr));

1504         mdb_printf("%-?p %-17s ", addr, macaddr);
1505         if (bfp->bf_flags & BFF_LOCALADDR)
1506                 mdb_printf("%-7s", "[self]");
1507         else
1508                 mdb_printf("t-%-5d", args->lbolt - bfp->bf_lastheard);
1509         mdb_printf(" %-7u ", bfp->bf_refs);

1511         if (bfp->bf_trill_nick != 0) {
1512                 mdb_printf("%d\n", bfp->bf_trill_nick);
1513         } else {
```



```
1514                 for (i = 0; i < bfp->bf_nlinks; i++) {
1515                         print_link_name(args, (uintptr_t)links[i],
1516                             i == bfp->bf_nlinks - 1 ? '\n' : ' ');
1517                 }
1518         }

1520         return (WALK_NEXT);
1521 }

1523 static int
1524 do_show_bridge(uintptr_t addr, const void *data, void *ptr)
1525 {
1526         show_bridge_args_t *args = ptr;
1527         bridge_inst_t bi;
1528         const bridge_inst_t *bip;
1529         trill_node_t tn;
1530         trill_sock_t tsp;
1531         trill_nickinfo_t tni;
1532         char bname[MAXLINKNAMELEN];
1533         char macaddr[ETHERADDRL * 3];
1534         char *cp;
1535         uint_t nnicks;
1536         int i;

1538         if (data != NULL) {
1539                 bip = data;
1540         } else {
1541                 if (mdb_vread(&bi, sizeof (bi), addr) == -1) {
1542                         mdb_warn("cannot read bridge instance at %p", addr);
1543                         return (WALK_ERR);
1544                 }
1545                 bip = &bi;
1546         }

1548         (void) strncpy(bname, bip->bi_name, sizeof (bname) - 1);
1549         bname[MAXLINKNAMELEN - 1] = '\0';
1550         cp = bname + strlen(bname);
1551         if (cp > bname && cp[-1] == '0')
1552                 cp[-1] = '\0';

1554         if (args->name != NULL && strcmp(args->name, bname) != 0)
1555                 return (WALK_NEXT);

1557         args->found = B_TRUE;
1558         args->nlinks = args->nfwd = 0;

1560         if (args->opt_l) {
1561                 mdb_printf("%-?s %-16s %-17s %3s %-4s ", "ADDR", "LINK",
1562                     "MAC-ADDR", "FLG", "PVID");
1563                 if (bip->bi_trilldata == NULL)
1564                         mdb_printf("%-5s %s\n", "STATE", "VLANS");
1565                 else
1566                         mdb_printf("%s\n", "FWD-VLANS");
1567         }

1569         if (!args->opt_f && !args->opt_t &&
1570             mdb_pwalk("list", do_bridge_links, args,
1571             addr + offsetof(bridge_inst_t, bi_links)) != DCMD_OK)
1572                 return (WALK_ERR);

1574         if (args->opt_f)
1575                 mdb_printf("%-?s %-17s %-7s %-7s %s\n", "ADDR", "DEST", "TIME",
1576                     "REFS", "OUTPUT");

1578         if (!args->opt_l && !args->opt_t &&
1579             mdb_pwalk("avl", do_bridge_fwd, args,
```

```
1580             addr + offsetof(bridge_inst_t, bi_fwd)) != DCMD_OK)
1581                 return (WALK_ERR);

1583         nnicks = 0;
1584         if (bip->bi_trilldata != NULL && !args->opt_l && !args->opt_f) {
1585                 if (mdb_vread(&args->ti, sizeof (args->ti),
1586                     (uintptr_t)bip->bi_trilldata) == -1) {
1587                         mdb_warn("cannot read trill instance at %p",
1588                             bip->bi_trilldata);
1589                         return (WALK_ERR);
1590                 }
1591                 if (args->opt_t)
1592                         mdb_printf("%-?s %-5s %-17s %s\n", "ADDR",
1593                             "NICK", "NEXT-HOP", "LINK");
1594                 for (i = 0; i < RBRIDGE_NICKNAME_MAX; i++) {
1595                         if (args->ti.ti_nodes[i] == NULL)
1596                                 continue;
1597                         if (args->opt_t) {
1598                                 if (mdb_vread(&tn, sizeof (tn),
1599                                     (uintptr_t)args->ti.ti_nodes[i]) == -1) {
1600                                         mdb_warn("cannot read trill node %d at "
1601                                             "%p", i, args->ti.ti_nodes[i]);
1602                                         return (WALK_ERR);
1603                                 }
1604                                 if (mdb_vread(&tni, sizeof (tni),
1605                                     (uintptr_t)tn.tn_ni) == -1) {
1606                                         mdb_warn("cannot read trill node info "
1607                                             "%d at %p", i, tn.tn_ni);
1608                                         return (WALK_ERR);
1609                                 }
1610                                 mdb_mac_addr(tni.tni_adjsnpa, ETHERADDRL,
1611                                     macaddr, sizeof (macaddr));
1612                                 if (tni.tni_nick == args->ti.ti_nick) {
1613                                         (void) strcpy(macaddr, "[self]");
1614                                 }
1615                                 mdb_printf("%-?p %-5u %-17s ",
1616                                     args->ti.ti_nodes[i], tni.tni_nick,
1617                                     macaddr);
1618                                 if (tn.tn_tsp != NULL) {
1619                                         if (mdb_vread(&tsp, sizeof (tsp),
1620                                             (uintptr_t)tn.tn_tsp) == -1) {
1621                                                 mdb_warn("cannot read trill "
1622                                                     "socket info at %p",
1623                                                     tn.tn_tsp);
1624                                                 return (WALK_ERR);
1625                                         }
1626                                         if (tsp.ts_link != NULL) {
1627                                                 print_link_name(args,
1628                                                     (uintptr_t)tsp.ts_link,
1629                                                     '\n');
1630                                                 continue;
1631                                         }
1632                                 }
1633                                 mdb_printf("--\n");
1634                         } else {
1635                                 nnicks++;
1636                         }
1637                 }
1638         } else {
1639                 if (args->opt_t)
1640                         mdb_printf("bridge is not running TRILL\n");
1641         }

1643         if (!args->opt_l && !args->opt_f && !args->opt_t) {
1644                 mdb_printf("%-?p %-7s %-16s %-7u %-7u", addr,
1645                     bip->bi_trilldata == NULL ? "stp" : "trill", bname,
```



```
1646                     args->nlinks, args->nfwd);
1647                 if (bip->bi_trilldata != NULL)
1648                         mdb_printf(" %-7u %u\n", nnicks, args->ti.ti_nick);
1649                 else
1650                         mdb_printf(" %-7s %s\n", "--", "--");
1651         }
1652         return (WALK_NEXT);
1653 }

1655 static int
1656 dladm_show_bridge(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1657 {
1658         show_bridge_args_t *args;
1659         GElf_Sym sym;
1660         int i;

1662         args = mdb_zalloc(sizeof (*args), UM_SLEEP);

1664         i = mdb_getopts(argc, argv,
1665             'l', MDB_OPT_SETBITS, 1, &args->opt_l,
1666             'f', MDB_OPT_SETBITS, 1, &args->opt_f,
1667             't', MDB_OPT_SETBITS, 1, &args->opt_t,
1668             NULL);

1670         argc -= i;
1671         argv += i;

1673         if (argc > 1 || (argc == 1 && argv[0].a_type != MDB_TYPE_STRING)) {
1674                 mdb_free(args, sizeof (*args));
1675                 return (DCMD_USAGE);
1676         }
1677         if (argc == 1)
1678                 args->name = argv[0].a_un.a_str;

1680         if ((args->lbolt = mdb_get_lbolt()) == -1) {
1681                 mdb_warn("failed to read lbolt");
1682                 goto err;
1683         }

1685         if (flags & DCMD_ADDRSPEC) {
1686                 if (args->name != NULL) {
1687                         mdb_printf("bridge name and address are mutually "
1688                             "exclusive\n");
1689                         goto err;
1690                 }
1691                 if (!args->opt_l && !args->opt_f && !args->opt_t)
1692                         mdb_printf("%-?s %-7s %-16s %-7s %-7s\n", "ADDR",
1693                             "PROTECT", "NAME", "NLINKS", "NFWD");
1694                 if (do_show_bridge(addr, NULL, args) != WALK_NEXT)
1695                         goto err;
1696                 mdb_free(args, sizeof (*args));
1697                 return (DCMD_OK);
1698         } else {
1699                 if ((args->opt_l || args->opt_f || args->opt_t) &&
1700                     args->name == NULL) {
1701                         mdb_printf("need bridge name or address with -[lft]\n");
1702                         goto err;
1703                 }
1704                 if (mdb_lookup_by_obj("bridge", "inst_list", &sym) == -1) {
1705                         mdb_warn("failed to find 'bridge'inst_list'");
1706                         goto err;
1707                 }
1708                 if (!args->opt_l && !args->opt_f && !args->opt_t)
1709                         mdb_printf("%-?s %-7s %-16s %-7s %-7s %-7s %s\n",
1710                             "ADDR", "PROTECT", "NAME", "NLINKS", "NFWD",
1711                             "NNICKS", "NICK");
```

```
1712                 if (mdb_pwalk("list", do_show_bridge, args,
1713                     (uintptr_t)sym.st_value) != DCMD_OK)
1714                         goto err;
1715                 if (!args->found && args->name != NULL) {
1716                         mdb_printf("bridge instance %s not found\n",
1717                             args->name);
1718                         goto err;
1719                 }
1720                 mdb_free(args, sizeof (*args));
1721                 return (DCMD_OK);
1722         }

1724 err:
1725         mdb_free(args, sizeof (*args));
1726         return (DCMD_ERR);
1727 }

1729 /*
1730  * Support for the "::dladm" dcmd
1731  */
1732 int
1733 dladm(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1734 {
1735         if (argc < 1 || argv[0].a_type != MDB_TYPE_STRING)
1736                 return (DCMD_USAGE);

1738         /*
1739          * This could be a bit more elaborate, once we support more of the
1740          * dladm show-* subcommands.
1741          */
1742         argc--;
1743         argv++;
1744         if (strcmp(argv[-1].a_un.a_str, "show-bridge") == 0)
1745                 return (dladm_show_bridge(addr, flags, argc, argv));

1747         return (DCMD_USAGE);
1748 }

1750 void
1751 dladm_help(void)
1752 {
1753         mdb_printf("Subcommands:\n"
1754             "  show-bridge [-flt] [<name>]\n"
1755             "\t     Show bridge information; -l for links and -f for "
1756             "forwarding\n"
1757             "\t     entries, and -t for TRILL nicknames.  Address is required "
1758             "if name\n"
1759             "\t     is not specified.\n");
1760 }
```

```
**********************************************************
    2239 Mon Jul  9 14:38:09 2012
new/usr/src/cmd/mdb/common/modules/genunix/net.h
dccp: build fixes, mdb (vfs sonode missing)
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _NET_H
#define _NET_H

#ifdef  __cplusplus
extern "C" {
#endif

extern struct mi_payload_walk_arg_s mi_icmp_arg;
extern struct mi_payload_walk_arg_s mi_ill_arg;

extern int sonode_walk_init(mdb_walk_state_t *);
extern int sonode_walk_step(mdb_walk_state_t *);
extern void sonode_walk_fini(mdb_walk_state_t *);
extern int mi_walk_init(mdb_walk_state_t *);
extern int mi_walk_step(mdb_walk_state_t *);
extern void mi_walk_fini(mdb_walk_state_t *);
extern int mi_payload_walk_init(mdb_walk_state_t *);
extern int mi_payload_walk_step(mdb_walk_state_t *);
extern int icmp_stacks_walk_init(mdb_walk_state_t *);
extern int icmp_stacks_walk_step(mdb_walk_state_t *);
extern int tcp_stacks_walk_init(mdb_walk_state_t *);
extern int tcp_stacks_walk_step(mdb_walk_state_t *);
extern int udp_stacks_walk_init(mdb_walk_state_t *);
extern int udp_stacks_walk_step(mdb_walk_state_t *);
extern int dccp_stacks_walk_init(mdb_walk_state_t *);
extern int dccp_stacks_walk_step(mdb_walk_state_t *);
#endif /* ! codereview */

extern int sonode(uintptr_t, uint_t, int, const mdb_arg_t *);
extern int mi(uintptr_t, uint_t, int, const mdb_arg_t *);
extern int netstat(uintptr_t, uint_t, int, const mdb_arg_t *);
extern int dladm(uintptr_t, uint_t, int, const mdb_arg_t *);
extern void dladm_help(void);

#ifdef  __cplusplus
}
```



```
#endif

#endif  /* _NET_H */
```

```
**********************************************************
   93952 Mon Jul  9 14:38:10 2012
new/usr/src/cmd/mdb/common/modules/ip/ip.c
dccp: conn_t
**********************************************************
______unchanged_portion_omitted_

 342 /*
 343  * Generic network stack walker initialization function.  It is used by all
 344  * other network stack walkers.
 345  * other netwrok stack walkers.
 346  */
 347 int
 348 ns_walk_init(mdb_walk_state_t *wsp)
 349 {
 350         if (mdb_layered_walk("netstack", wsp) == -1) {
 351                 mdb_warn("can't walk 'netstack'");
 352                 return (WALK_ERR);
 353         }
 354         return (WALK_NEXT);
 355 }
______unchanged_portion_omitted_

 377 /*
 378  * DCCP network stack walker stepping function.
 379  */
 380 int
 381 dccp_stacks_walk_step(mdb_walk_state_t *wsp)
 382 {
 383         return (ns_walk_step(wsp, NS_DCCP));
 384 }

 386 /*
 387 #endif /* ! codereview */
 388  * IP network stack walker stepping function.
 389  */
 390 int
 391 ip_stacks_walk_step(mdb_walk_state_t *wsp)
 392 {
 393         return (ns_walk_step(wsp, NS_IP));
 394 }

 396 /*
 397  * TCP network stack walker stepping function.
 398  */
 399 int
 400 tcp_stacks_walk_step(mdb_walk_state_t *wsp)
 401 {
 402         return (ns_walk_step(wsp, NS_TCP));
 403 }

 405 /*
 406  * SCTP network stack walker stepping function.
 407  */
 408 int
 409 sctp_stacks_walk_step(mdb_walk_state_t *wsp)
 410 {
 411         return (ns_walk_step(wsp, NS_SCTP));
 412 }

 414 /*
 415  * UDP network stack walker stepping function.
 416  */
 417 int
 418 udp_stacks_walk_step(mdb_walk_state_t *wsp)
 419 {
```



```
 419         return (ns_walk_step(wsp, NS_UDP));
 420 }

 422 /*
 423  * Initialization function for the per CPU TCP stats counter walker of a given
 424  * TCP stack.
 425  */
 426 int
 427 tcps_sc_walk_init(mdb_walk_state_t *wsp)
 428 {
 429         tcp_stack_t tcps;

 431         if (wsp->walk_addr == NULL)
 432                 return (WALK_ERR);

 434         if (mdb_vread(&tcps, sizeof (tcps), wsp->walk_addr) == -1) {
 435                 mdb_warn("failed to read tcp_stack_t at %p", wsp->walk_addr);
 436                 return (WALK_ERR);
 437         }
 438         if (tcps.tcps_sc_cnt == 0)
 439                 return (WALK_DONE);

 441         /*
 442          * Store the tcp_stack_t pointer in walk_data.  The stepping function
 443          * used it to calculate if the end of the counter has reached.
 444          */
 445         wsp->walk_data = (void *)wsp->walk_addr;
 446         wsp->walk_addr = (uintptr_t)tcps.tcps_sc;
 447         return (WALK_NEXT);
 448 }

 450 /*
 451  * Stepping function for the per CPU TCP stats counterwalker.
 452  */
 453 int
 454 tcps_sc_walk_step(mdb_walk_state_t *wsp)
 455 {
 456         int status;
 457         tcp_stack_t tcps;
 458         tcp_stats_cpu_t *stats;
 459         char *next, *end;

 461         if (mdb_vread(&tcps, sizeof (tcps), (uintptr_t)wsp->walk_data) == -1) {
 462                 mdb_warn("failed to read tcp_stack_t at %p", wsp->walk_addr);
 463                 return (WALK_ERR);
 464         }
 465         if (mdb_vread(&stats, sizeof (stats), wsp->walk_addr) == -1) {
 466                 mdb_warn("failed ot read tcp_stats_cpu_t at %p",
 467                     wsp->walk_addr);
 468                 return (WALK_ERR);
 469         }
 470         status = wsp->walk_callback((uintptr_t)stats, &stats, wsp->walk_cbdata);
 471         if (status != WALK_NEXT)
 472                 return (status);

 474         next = (char *)wsp->walk_addr + sizeof (tcp_stats_cpu_t *);
 475         end = (char *)tcps.tcps_sc + tcps.tcps_sc_cnt *
 476             sizeof (tcp_stats_cpu_t *);
 477         if (next >= end)
 478                 return (WALK_DONE);
 479         wsp->walk_addr = (uintptr_t)next;
 480         return (WALK_NEXT);
 481 }

 483 int
 484 th_hash_walk_init(mdb_walk_state_t *wsp)
```

```
{
        GElf_Sym sym;
        list_node_t *next;

        if (wsp->walk_addr == NULL) {
                if (mdb_lookup_by_obj("ip", "ip_thread_list", &sym) == 0) {
                        wsp->walk_addr = sym.st_value;
                } else {
                        mdb_warn("unable to locate ip_thread_list\n");
                        return (WALK_ERR);
                }
        }

        if (mdb_vread(&next, sizeof (next),
            wsp->walk_addr + offsetof(list_t, list_head) +
            offsetof(list_node_t, list_next)) == -1 ||
            next == NULL) {
                mdb_warn("non-DEBUG image; cannot walk th_hash list\n");
                return (WALK_ERR);
        }

        if (mdb_layered_walk("list", wsp) == -1) {
                mdb_warn("can't walk 'list'");
                return (WALK_ERR);
        } else {
                return (WALK_NEXT);
        }
}

int
th_hash_walk_step(mdb_walk_state_t *wsp)
{
        return (wsp->walk_callback(wsp->walk_addr, wsp->walk_layer,
            wsp->walk_cbdata));
}

/*
 * Called with walk_addr being the address of ips_ill_g_heads
 */
int
illif_stack_walk_init(mdb_walk_state_t *wsp)
{
        illif_walk_data_t *iw;

        if (wsp->walk_addr == NULL) {
                mdb_warn("illif_stack supports only local walks\n");
                return (WALK_ERR);
        }

        iw = mdb_alloc(sizeof (illif_walk_data_t), UM_SLEEP);

        if (mdb_vread(iw->ill_g_heads, MAX_G_HEADS * sizeof (ill_g_head_t),
            wsp->walk_addr) == -1) {
                mdb_warn("failed to read 'ips_ill_g_heads' at %p",
                    wsp->walk_addr);
                mdb_free(iw, sizeof (illif_walk_data_t));
                return (WALK_ERR);
        }

        iw->ill_list = 0;
        wsp->walk_addr = (uintptr_t)iw->ill_g_heads[0].ill_g_list_head;
        wsp->walk_data = iw;

        return (WALK_NEXT);
}
```



```
int
illif_stack_walk_step(mdb_walk_state_t *wsp)
{
        uintptr_t addr = wsp->walk_addr;
        illif_walk_data_t *iw = wsp->walk_data;
        int list = iw->ill_list;

        if (mdb_vread(&iw->ill_if, sizeof (ill_if_t), addr) == -1) {
                mdb_warn("failed to read ill_if_t at %p", addr);
                return (WALK_ERR);
        }

        wsp->walk_addr = (uintptr_t)iw->ill_if.illif_next;

        if (wsp->walk_addr ==
            (uintptr_t)iw->ill_g_heads[list].ill_g_list_head) {

                if (++list >= MAX_G_HEADS)
                        return (WALK_DONE);

                iw->ill_list = list;
                wsp->walk_addr =
                    (uintptr_t)iw->ill_g_heads[list].ill_g_list_head;
                return (WALK_NEXT);
        }

        return (wsp->walk_callback(addr, iw, wsp->walk_cbdata));
}

void
illif_stack_walk_fini(mdb_walk_state_t *wsp)
{
        mdb_free(wsp->walk_data, sizeof (illif_walk_data_t));
}

typedef struct illif_cbdata {
        uint_t ill_flags;
        uintptr_t ill_addr;
        int ill_printlist;      /* list to be printed (MAX_G_HEADS for all) */
        boolean_t ill_printed;
} illif_cbdata_t;

static int
illif_cb(uintptr_t addr, const illif_walk_data_t *iw, illif_cbdata_t *id)
{
        const char *version;

        if (id->ill_printlist < MAX_G_HEADS &&
            id->ill_printlist != iw->ill_list)
                return (WALK_NEXT);

        if (id->ill_flags & DCMD_ADDRSPEC && id->ill_addr != addr)
                return (WALK_NEXT);

        if (id->ill_flags & DCMD_PIPE_OUT) {
                mdb_printf("%p\n", addr);
                return (WALK_NEXT);
        }

        switch (iw->ill_list) {
                case IP_V4_G_HEAD:      version = "v4"; break;
                case IP_V6_G_HEAD:      version = "v6"; break;
                default:                version = "??"; break;
        }

        mdb_printf("%?p %2s %?p %10d %?p %s\n",
```

```
617             addr, version, addr + offsetof(ill_if_t, illif_avl_by_ppa),
618             iw->ill_if.illif_avl_by_ppa.avl_numnodes,
619             iw->ill_if.illif_ppa_arena, iw->ill_if.illif_name);

621         id->ill_printed = TRUE;

623         return (WALK_NEXT);
624 }

626 int
627 ip_stacks_common_walk_init(mdb_walk_state_t *wsp)
628 {
629         if (mdb_layered_walk("ip_stacks", wsp) == -1) {
630                 mdb_warn("can't walk 'ip_stacks'");
631                 return (WALK_ERR);
632         }

634         return (WALK_NEXT);
635 }

637 int
638 illif_walk_step(mdb_walk_state_t *wsp)
639 {
640         uintptr_t kaddr;

642         kaddr = wsp->walk_addr + OFFSETOF(ip_stack_t, ips_ill_g_heads);

644         if (mdb_vread(&kaddr, sizeof (kaddr), kaddr) == -1) {
645                 mdb_warn("can't read ips_ip_cache_table at %p", kaddr);
646                 return (WALK_ERR);
647         }

649         if (mdb_pwalk("illif_stack", wsp->walk_callback,
650             wsp->walk_cbdata, kaddr) == -1) {
651                 mdb_warn("couldn't walk 'illif_stack' for ips_ill_g_heads %p",
652                     kaddr);
653                 return (WALK_ERR);
654         }
655         return (WALK_NEXT);
656 }

658 int
659 illif(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
660 {
661         illif_cbdata_t id;
662         ill_if_t ill_if;
663         const char *opt_P = NULL;
664         int printlist = MAX_G_HEADS;

666         if (mdb_getopts(argc, argv,
667             'P', MDB_OPT_STR, &opt_P, NULL) != argc)
668                 return (DCMD_USAGE);

670         if (opt_P != NULL) {
671                 if (strcmp("v4", opt_P) == 0) {
672                         printlist = IP_V4_G_HEAD;
673                 } else if (strcmp("v6", opt_P) == 0) {
674                         printlist = IP_V6_G_HEAD;
675                 } else {
676                         mdb_warn("invalid protocol '%s'\n", opt_P);
677                         return (DCMD_USAGE);
678                 }
679         }

681         if (DCMD_HDRSPEC(flags) && (flags & DCMD_PIPE_OUT) == 0) {
682                 mdb_printf("%<u>%?s %2s %?s %10s %?s %-10s%</u>\n",
```



```
683                     "ADDR", "IP", "AVLADDR", "NUMNODES", "ARENA", "NAME");
684         }

686         id.ill_flags = flags;
687         id.ill_addr = addr;
688         id.ill_printlist = printlist;
689         id.ill_printed = FALSE;

691         if (mdb_walk("illif", (mdb_walk_cb_t)illif_cb, &id) == -1) {
692                 mdb_warn("can't walk ill_if_t structures");
693                 return (DCMD_ERR);
694         }

696         if (!(flags & DCMD_ADDRSPEC) || opt_P != NULL || id.ill_printed)
697                 return (DCMD_OK);

699         /*
700          * If an address is specified and the walk doesn't find it,
701          * print it anyway.
702          */
703         if (mdb_vread(&ill_if, sizeof (ill_if_t), addr) == -1) {
704                 mdb_warn("failed to read ill_if_t at %p", addr);
705                 return (DCMD_ERR);
706         }

708         mdb_printf("%?p %2s %?p %10d %?p %s\n",
709             addr, "??", addr + offsetof(ill_if_t, illif_avl_by_ppa),
710             ill_if.illif_avl_by_ppa.avl_numnodes,
711             ill_if.illif_ppa_arena, ill_if.illif_name);

713         return (DCMD_OK);
714 }

716 static void
717 illif_help(void)
718 {
719         mdb_printf("Options:\n");
720         mdb_printf("\t-P v4 | v6"
721             "\tfilter interface structures for the specified protocol\n");
722 }

724 int
725 nce_walk_init(mdb_walk_state_t *wsp)
726 {
727         if (mdb_layered_walk("nce_cache", wsp) == -1) {
728                 mdb_warn("can't walk 'nce_cache'");
729                 return (WALK_ERR);
730         }

732         return (WALK_NEXT);
733 }

735 int
736 nce_walk_step(mdb_walk_state_t *wsp)
737 {
738         nce_t nce;

740         if (mdb_vread(&nce, sizeof (nce), wsp->walk_addr) == -1) {
741                 mdb_warn("can't read nce at %p", wsp->walk_addr);
742                 return (WALK_ERR);
743         }

745         return (wsp->walk_callback(wsp->walk_addr, &nce, wsp->walk_cbdata));
746 }

748 static int
```

```
749 nce_format(uintptr_t addr, const nce_t *ncep, void *nce_cb_arg)
750 {
751         nce_cbdata_t *nce_cb = nce_cb_arg;
752         ill_t ill;
753         char ill_name[LIFNAMSIZ];
754         ncec_t ncec;

756         if (mdb_vread(&ncec, sizeof (ncec),
757             (uintptr_t)ncep->nce_common) == -1) {
758                 mdb_warn("can't read ncec at %p", ncep->nce_common);
759                 return (WALK_NEXT);
760         }
761         if (nce_cb->nce_ipversion != 0 &&
762             ncec.ncec_ipversion != nce_cb->nce_ipversion)
763                 return (WALK_NEXT);

765         if (mdb_vread(&ill, sizeof (ill), (uintptr_t)ncep->nce_ill) == -1) {
766                 mdb_snprintf(ill_name, sizeof (ill_name), "--");
767         } else {
768                 (void) mdb_readstr(ill_name,
769                     MIN(LIFNAMSIZ, ill.ill_name_length),
770                     (uintptr_t)ill.ill_name);
771         }

773         if (nce_cb->nce_ill_name[0] != '\0' &&
774             strncmp(nce_cb->nce_ill_name, ill_name, LIFNAMSIZ) != 0)
775                 return (WALK_NEXT);

777         if (ncec.ncec_ipversion == IPV6_VERSION) {

779                 mdb_printf("%?p %5s %-18s %?p %6d %N\n",
780                     addr, ill_name,
781                     nce_l2_addr(ncep, &ill),
782                     ncep->nce_fp_mp,
783                     ncep->nce_refcnt,
784                     &ncep->nce_addr);

786         } else {
787                 struct in_addr nceaddr;

789                 IN6_V4MAPPED_TO_INADDR(&ncep->nce_addr, &nceaddr);
790                 mdb_printf("%?p %5s %-18s %?p %6d %I\n",
791                     addr, ill_name,
792                     nce_l2_addr(ncep, &ill),
793                     ncep->nce_fp_mp,
794                     ncep->nce_refcnt,
795                     nceaddr.s_addr);
796         }

798         return (WALK_NEXT);
799 }

801 int
802 dce_walk_init(mdb_walk_state_t *wsp)
803 {
804         wsp->walk_data = (void *)wsp->walk_addr;

806         if (mdb_layered_walk("dce_cache", wsp) == -1) {
807                 mdb_warn("can't walk 'dce_cache'");
808                 return (WALK_ERR);
809         }

811         return (WALK_NEXT);
812 }

814 int
```



```
815 dce_walk_step(mdb_walk_state_t *wsp)
816 {
817         dce_t dce;

819         if (mdb_vread(&dce, sizeof (dce), wsp->walk_addr) == -1) {
820                 mdb_warn("can't read dce at %p", wsp->walk_addr);
821                 return (WALK_ERR);
822         }

824         /* If ip_stack_t is specified, skip DCEs that don't belong to it. */
825         if ((wsp->walk_data != NULL) && (wsp->walk_data != dce.dce_ipst))
826                 return (WALK_NEXT);

828         return (wsp->walk_callback(wsp->walk_addr, &dce, wsp->walk_cbdata));
829 }

831 int
832 ire_walk_init(mdb_walk_state_t *wsp)
833 {
834         wsp->walk_data = (void *)wsp->walk_addr;

836         if (mdb_layered_walk("ire_cache", wsp) == -1) {
837                 mdb_warn("can't walk 'ire_cache'");
838                 return (WALK_ERR);
839         }

841         return (WALK_NEXT);
842 }

844 int
845 ire_walk_step(mdb_walk_state_t *wsp)
846 {
847         ire_t ire;

849         if (mdb_vread(&ire, sizeof (ire), wsp->walk_addr) == -1) {
850                 mdb_warn("can't read ire at %p", wsp->walk_addr);
851                 return (WALK_ERR);
852         }

854         /* If ip_stack_t is specified, skip IREs that don't belong to it. */
855         if ((wsp->walk_data != NULL) && (wsp->walk_data != ire.ire_ipst))
856                 return (WALK_NEXT);

858         return (wsp->walk_callback(wsp->walk_addr, &ire, wsp->walk_cbdata));
859 }

861 /* ARGSUSED */
862 int
863 ire_next_walk_init(mdb_walk_state_t *wsp)
864 {
865         return (WALK_NEXT);
866 }

868 int
869 ire_next_walk_step(mdb_walk_state_t *wsp)
870 {
871         ire_t ire;
872         int status;


875         if (wsp->walk_addr == NULL)
876                 return (WALK_DONE);

878         if (mdb_vread(&ire, sizeof (ire), wsp->walk_addr) == -1) {
879                 mdb_warn("can't read ire at %p", wsp->walk_addr);
880                 return (WALK_ERR);
```

```
881         }
882         status = wsp->walk_callback(wsp->walk_addr, &ire,
883             wsp->walk_cbdata);
885         if (status != WALK_NEXT)
886                 return (status);
888         wsp->walk_addr = (uintptr_t)ire.ire_next;
889         return (status);
890 }
892 static int
893 ire_format(uintptr_t addr, const void *ire_arg, void *ire_cb_arg)
894 {
895         const ire_t *irep = ire_arg;
896         ire_cbdata_t *ire_cb = ire_cb_arg;
897         boolean_t verbose = ire_cb->verbose;
898         ill_t ill;
899         char ill_name[LIFNAMSIZ];
900         boolean_t condemned = irep->ire_generation == IRE_GENERATION_CONDEMNED;
902         static const mdb_bitmask_t tmasks[] = {
903                 { "BROADCAST",  IRE_BROADCAST,          IRE_BROADCAST   },
904                 { "DEFAULT",    IRE_DEFAULT,            IRE_DEFAULT     },
905                 { "LOCAL",      IRE_LOCAL,              IRE_LOCAL       },
906                 { "LOOPBACK",   IRE_LOOPBACK,           IRE_LOOPBACK    },
907                 { "PREFIX",     IRE_PREFIX,             IRE_PREFIX      },
908                 { "MULTICAST",  IRE_MULTICAST,          IRE_MULTICAST   },
909                 { "NOROUTE",    IRE_NOROUTE,            IRE_NOROUTE     },
910                 { "IF_NORESOLVER", IRE_IF_NORESOLVER,   IRE_IF_NORESOLVER },
911                 { "IF_RESOLVER", IRE_IF_RESOLVER,       IRE_IF_RESOLVER },
912                 { "IF_CLONE",   IRE_IF_CLONE,           IRE_IF_CLONE    },
913                 { "HOST",       IRE_HOST,               IRE_HOST        },
914                 { NULL,         0,                      0               }
915         };
917         static const mdb_bitmask_t fmasks[] = {
918                 { "UP",         RTF_UP,                 RTF_UP          },
919                 { "GATEWAY",    RTF_GATEWAY,            RTF_GATEWAY     },
920                 { "HOST",       RTF_HOST,               RTF_HOST        },
921                 { "REJECT",     RTF_REJECT,             RTF_REJECT      },
922                 { "DYNAMIC",    RTF_DYNAMIC,            RTF_DYNAMIC     },
923                 { "MODIFIED",   RTF_MODIFIED,           RTF_MODIFIED    },
924                 { "DONE",       RTF_DONE,               RTF_DONE        },
925                 { "MASK",       RTF_MASK,               RTF_MASK        },
926                 { "CLONING",    RTF_CLONING,            RTF_CLONING     },
927                 { "XRESOLVE",   RTF_XRESOLVE,           RTF_XRESOLVE    },
928                 { "LLINFO",     RTF_LLINFO,             RTF_LLINFO      },
929                 { "STATIC",     RTF_STATIC,             RTF_STATIC      },
930                 { "BLACKHOLE",  RTF_BLACKHOLE,          RTF_BLACKHOLE   },
931                 { "PRIVATE",    RTF_PRIVATE,            RTF_PRIVATE     },
932                 { "PROTO2",     RTF_PROTO2,             RTF_PROTO2      },
933                 { "PROTO1",     RTF_PROTO1,             RTF_PROTO1      },
934                 { "MULTIRT",    RTF_MULTIRT,            RTF_MULTIRT     },
935                 { "SETSRC",     RTF_SETSRC,             RTF_SETSRC      },
936                 { "INDIRECT",   RTF_INDIRECT,           RTF_INDIRECT    },
937                 { NULL,         0,                      0               }
938         };
940         if (ire_cb->ire_ipversion != 0 &&
941             irep->ire_ipversion != ire_cb->ire_ipversion)
942                 return (WALK_NEXT);
944         if (mdb_vread(&ill, sizeof (ill), (uintptr_t)irep->ire_ill) == -1) {
945                 mdb_snprintf(ill_name, sizeof (ill_name), "--");
946         } else {
```



```
947                 (void) mdb_readstr(ill_name,
948                     MIN(LIFNAMSIZ, ill.ill_name_length),
949                     (uintptr_t)ill.ill_name);
950         }
952         if (irep->ire_ipversion == IPV6_VERSION && verbose) {
954                 mdb_printf("%<b>%?p%</b>%3s %40N <%hb%s>\n"
955                     "%?s %40N\n"
956                     "%?s %40d %4d <%hb> %s\n",
957                     addr, condemned ? "(C)" : "", &irep->ire_setsrc_addr_v6,
958                     irep->ire_type, tmasks,
959                     (irep->ire_testhidden ? ", HIDDEN" : ""),
960                     "", &irep->ire_addr_v6,
961                     "", ips_to_stackid((uintptr_t)irep->ire_ipst),
962                     irep->ire_zoneid,
963                     irep->ire_flags, fmasks, ill_name);
965         } else if (irep->ire_ipversion == IPV6_VERSION) {
967                 mdb_printf("%?p%3s %30N %30N %5d %4d %s\n",
968                     addr, condemned ? "(C)" : "", &irep->ire_setsrc_addr_v6,
969                     &irep->ire_addr_v6,
970                     ips_to_stackid((uintptr_t)irep->ire_ipst),
971                     irep->ire_zoneid, ill_name);
973         } else if (verbose) {
975                 mdb_printf("%<b>%?p%</b>%3s %40I <%hb%s>\n"
976                     "%?s %40I\n"
977                     "%?s %40d %4d <%hb> %s\n",
978                     addr, condemned ? "(C)" : "", irep->ire_setsrc_addr,
979                     irep->ire_type, tmasks,
980                     (irep->ire_testhidden ? ", HIDDEN" : ""),
981                     "", irep->ire_addr,
982                     "", ips_to_stackid((uintptr_t)irep->ire_ipst),
983                     irep->ire_zoneid, irep->ire_flags, fmasks, ill_name);
985         } else {
987                 mdb_printf("%?p%3s %30I %30I %5d %4d %s\n", addr,
988                     condemned ? "(C)" : "", irep->ire_setsrc_addr,
989                     irep->ire_addr, ips_to_stackid((uintptr_t)irep->ire_ipst),
990                     irep->ire_zoneid, ill_name);
991         }
993         return (WALK_NEXT);
994 }
996 /*
997  * There are faster ways to do this.  Given the interactive nature of this
998  * use I don't think its worth much effort.
999  */
1000 static unsigned short
1001 ipcksum(void *p, int len)
1002 {
1003         int32_t sum = 0;
1005         while (len > 1) {
1006                 /* alignment */
1007                 sum += *(uint16_t *)p;
1008                 p = (char *)p + sizeof (uint16_t);
1009                 if (sum & 0x80000000)
1010                         sum = (sum & 0xFFFF) + (sum >> 16);
1011                 len -= 2;
1012         }
```

```
1014         if (len)
1015                 sum += (uint16_t)*(unsigned char *)p;
1016
1017         while (sum >> 16)
1018                 sum = (sum & 0xFFFF) + (sum >> 16);
1019
1020         return (~sum);
1021 }
1022
1023 static const mdb_bitmask_t tcp_flags[] = {
1024         { "SYN",        TH_SYN,         TH_SYN  },
1025         { "ACK",        TH_ACK,         TH_ACK  },
1026         { "FIN",        TH_FIN,         TH_FIN  },
1027         { "RST",        TH_RST,         TH_RST  },
1028         { "PSH",        TH_PUSH,        TH_PUSH },
1029         { "ECE",        TH_ECE,         TH_ECE  },
1030         { "CWR",        TH_CWR,         TH_CWR  },
1031         { NULL,         0,              0       }
1032 };
1033
1034 /* TCP option length */
1035 #define TCPOPT_HEADER_LEN       2
1036 #define TCPOPT_MAXSEG_LEN       4
1037 #define TCPOPT_WS_LEN           3
1038 #define TCPOPT_TSTAMP_LEN       10
1039 #define TCPOPT_SACK_OK_LEN      2
1040
1041 static void
1042 tcphdr_print_options(uint8_t *opts, uint32_t opts_len)
1043 {
1044         uint8_t *endp;
1045         uint32_t len, val;
1046
1047         mdb_printf("%<b>Options:%</b>");
1048         endp = opts + opts_len;
1049         while (opts < endp) {
1050                 len = endp - opts;
1051                 switch (*opts) {
1052                 case TCPOPT_EOL:
1053                         mdb_printf(" EOL");
1054                         opts++;
1055                         break;
1056
1057                 case TCPOPT_NOP:
1058                         mdb_printf(" NOP");
1059                         opts++;
1060                         break;
1061
1062                 case TCPOPT_MAXSEG: {
1063                         uint16_t mss;
1064
1065                         if (len < TCPOPT_MAXSEG_LEN ||
1066                             opts[1] != TCPOPT_MAXSEG_LEN) {
1067                                 mdb_printf(" <Truncated MSS>\n");
1068                                 return;
1069                         }
1070                         mdb_nhconvert(&mss, opts + TCPOPT_HEADER_LEN,
1071                             sizeof (mss));
1072                         mdb_printf(" MSS=%u", mss);
1073                         opts += TCPOPT_MAXSEG_LEN;
1074                         break;
1075                 }
1076
1077                 case TCPOPT_WSCALE:
1078                         if (len < TCPOPT_WS_LEN || opts[1] != TCPOPT_WS_LEN) {
```



```
1079                                 mdb_printf(" <Truncated WS>\n");
1080                                 return;
1081                         }
1082                         mdb_printf(" WS=%u", opts[2]);
1083                         opts += TCPOPT_WS_LEN;
1084                         break;
1085
1086                 case TCPOPT_TSTAMP: {
1087                         if (len < TCPOPT_TSTAMP_LEN ||
1088                             opts[1] != TCPOPT_TSTAMP_LEN) {
1089                                 mdb_printf(" <Truncated TS>\n");
1090                                 return;
1091                         }
1092
1093                         opts += TCPOPT_HEADER_LEN;
1094                         mdb_nhconvert(&val, opts, sizeof (val));
1095                         mdb_printf(" TS_VAL=%u,", val);
1096
1097                         opts += sizeof (val);
1098                         mdb_nhconvert(&val, opts, sizeof (val));
1099                         mdb_printf("TS_ECHO=%u", val);
1100
1101                         opts += sizeof (val);
1102                         break;
1103                 }
1104
1105                 case TCPOPT_SACK_PERMITTED:
1106                         if (len < TCPOPT_SACK_OK_LEN ||
1107                             opts[1] != TCPOPT_SACK_OK_LEN) {
1108                                 mdb_printf(" <Truncated SACK_OK>\n");
1109                                 return;
1110                         }
1111                         mdb_printf(" SACK_OK");
1112                         opts += TCPOPT_SACK_OK_LEN;
1113                         break;
1114
1115                 case TCPOPT_SACK: {
1116                         uint32_t sack_len;
1117
1118                         if (len <= TCPOPT_HEADER_LEN || len < opts[1] ||
1119                             opts[1] <= TCPOPT_HEADER_LEN) {
1120                                 mdb_printf(" <Truncated SACK>\n");
1121                                 return;
1122                         }
1123                         sack_len = opts[1] - TCPOPT_HEADER_LEN;
1124                         opts += TCPOPT_HEADER_LEN;
1125
1126                         mdb_printf(" SACK=");
1127                         while (sack_len > 0) {
1128                                 if (opts + 2 * sizeof (val) > endp) {
1129                                         mdb_printf("<Truncated SACK>\n");
1130                                         opts = endp;
1131                                         break;
1132                                 }
1133
1134                                 mdb_nhconvert(&val, opts, sizeof (val));
1135                                 mdb_printf("<%u,", val);
1136                                 opts += sizeof (val);
1137                                 mdb_nhconvert(&val, opts, sizeof (val));
1138                                 mdb_printf("%u>", val);
1139                                 opts += sizeof (val);
1140
1141                                 sack_len -= 2 * sizeof (val);
1142                         }
1143                         break;
1144                 }
```

```
1146                 default:
1147                         mdb_printf(" Opts=<val=%u,len=%u>", *opts,
1148                             opts[1]);
1149                         opts += opts[1];
1150                         break;
1151                 }
1152         }
1153         mdb_printf("\n");
1154 }

1156 static void
1157 tcphdr_print(struct tcphdr *tcph)
1158 {
1159         in_port_t       sport, dport;
1160         tcp_seq         seq, ack;
1161         uint16_t        win, urp;

1163         mdb_printf("%<b>TCP header%</b>\n");

1165         mdb_nhconvert(&sport, &tcph->th_sport, sizeof (sport));
1166         mdb_nhconvert(&dport, &tcph->th_dport, sizeof (dport));
1167         mdb_nhconvert(&seq, &tcph->th_seq, sizeof (seq));
1168         mdb_nhconvert(&ack, &tcph->th_ack, sizeof (ack));
1169         mdb_nhconvert(&win, &tcph->th_win, sizeof (win));
1170         mdb_nhconvert(&urp, &tcph->th_urp, sizeof (urp));

1172         mdb_printf("%<u>%6s %6s %10s %10s %4s %5s %5s %5s %-15s%</u>\n",
1173             "SPORT", "DPORT", "SEQ", "ACK", "HLEN", "WIN", "CSUM", "URP",
1174             "FLAGS");
1175         mdb_printf("%6hu %6hu %10u %10u %4d %5hu %5hu %5hu <%b>\n",
1176             sport, dport, seq, ack, tcph->th_off << 2, win,
1177             tcph->th_sum, urp, tcph->th_flags, tcp_flags);
1178         mdb_printf("0x%04x 0x%04x 0x%08x 0x%08x\n\n",
1179             sport, dport, seq, ack);
1180 }

1182 /* ARGSUSED */
1183 static int
1184 tcphdr(uintptr_t addr, uint_t flags, int ac, const mdb_arg_t *av)
1185 {
1186         struct tcphdr   tcph;
1187         uint32_t        opt_len;

1189         if (!(flags & DCMD_ADDRSPEC))
1190                 return (DCMD_USAGE);

1192         if (mdb_vread(&tcph, sizeof (tcph), addr) == -1) {
1193                 mdb_warn("failed to read TCP header at %p", addr);
1194                 return (DCMD_ERR);
1195         }
1196         tcphdr_print(&tcph);

1198         /* If there are options, print them out also. */
1199         opt_len = (tcph.th_off << 2) - TCP_MIN_HEADER_LENGTH;
1200         if (opt_len > 0) {
1201                 uint8_t *opts, *opt_buf;

1203                 opt_buf = mdb_alloc(opt_len, UM_SLEEP);
1204                 opts = (uint8_t *)addr + sizeof (tcph);
1205                 if (mdb_vread(opt_buf, opt_len, (uintptr_t)opts) == -1) {
1206                         mdb_warn("failed to read TCP options at %p", opts);
1207                         return (DCMD_ERR);
1208                 }
1209                 tcphdr_print_options(opt_buf, opt_len);
1210                 mdb_free(opt_buf, opt_len);
```



```
1211         }

1213         return (DCMD_OK);
1214 }

1216 static void
1217 udphdr_print(struct udphdr *udph)
1218 {
1219         in_port_t       sport, dport;
1220         uint16_t        hlen;

1222         mdb_printf("%<b>UDP header%</b>\n");

1224         mdb_nhconvert(&sport, &udph->uh_sport, sizeof (sport));
1225         mdb_nhconvert(&dport, &udph->uh_dport, sizeof (dport));
1226         mdb_nhconvert(&hlen, &udph->uh_ulen, sizeof (hlen));

1228         mdb_printf("%<u>%14s %14s %5s %6s%</u>\n",
1229             "SPORT", "DPORT", "LEN", "CSUM");
1230         mdb_printf("%5hu (0x%04x) %5hu (0x%04x) %5hu 0x%04hx\n\n", sport, sport,
1231             dport, dport, hlen, udph->uh_sum);
1232 }

1234 /* ARGSUSED */
1235 static int
1236 udphdr(uintptr_t addr, uint_t flags, int ac, const mdb_arg_t *av)
1237 {
1238         struct udphdr   udph;

1240         if (!(flags & DCMD_ADDRSPEC))
1241                 return (DCMD_USAGE);

1243         if (mdb_vread(&udph, sizeof (udph), addr) == -1) {
1244                 mdb_warn("failed to read UDP header at %p", addr);
1245                 return (DCMD_ERR);
1246         }
1247         udphdr_print(&udph);
1248         return (DCMD_OK);
1249 }

1251 static void
1252 sctphdr_print(sctp_hdr_t *sctph)
1253 {
1254         in_port_t sport, dport;

1256         mdb_printf("%<b>SCTP header%</b>\n");
1257         mdb_nhconvert(&sport, &sctph->sh_sport, sizeof (sport));
1258         mdb_nhconvert(&dport, &sctph->sh_dport, sizeof (dport));

1260         mdb_printf("%<u>%14s %14s %10s %10s%</u>\n",
1261             "SPORT", "DPORT", "VTAG", "CHKSUM");
1262         mdb_printf("%5hu (0x%04x) %5hu (0x%04x) %10u 0x%08x\n\n", sport, sport,
1263             dport, dport, sctph->sh_verf, sctph->sh_chksum);
1264 }

1266 /* ARGSUSED */
1267 static int
1268 sctphdr(uintptr_t addr, uint_t flags, int ac, const mdb_arg_t *av)
1269 {
1270         sctp_hdr_t sctph;

1272         if (!(flags & DCMD_ADDRSPEC))
1273                 return (DCMD_USAGE);

1275         if (mdb_vread(&sctph, sizeof (sctph), addr) == -1) {
1276                 mdb_warn("failed to read SCTP header at %p", addr);
```

```
1277                return (DCMD_ERR);
1278        }

1280        sctphdr_print(&sctph);
1281        return (DCMD_OK);
1282 }

1284 static int
1285 transport_hdr(int proto, uintptr_t addr)
1286 {
1287        mdb_printf("\n");
1288        switch (proto) {
1289        case IPPROTO_TCP: {
1290                struct tcphdr tcph;

1292                if (mdb_vread(&tcph, sizeof (tcph), addr) == -1) {
1293                        mdb_warn("failed to read TCP header at %p", addr);
1294                        return (DCMD_ERR);
1295                }
1296                tcphdr_print(&tcph);
1297                break;
1298        }
1299        case IPPROTO_UDP:  {
1300                struct udphdr udph;

1302                if (mdb_vread(&udph, sizeof (udph), addr) == -1) {
1303                        mdb_warn("failed to read UDP header at %p", addr);
1304                        return (DCMD_ERR);
1305                }
1306                udphdr_print(&udph);
1307                break;
1308        }
1309        case IPPROTO_SCTP: {
1310                sctp_hdr_t sctph;

1312                if (mdb_vread(&sctph, sizeof (sctph), addr) == -1) {
1313                        mdb_warn("failed to read SCTP header at %p", addr);
1314                        return (DCMD_ERR);
1315                }
1316                sctphdr_print(&sctph);
1317                break;
1318        }
1319        default:
1320                break;
1321        }

1323        return (DCMD_OK);
1324 }

1326 static const mdb_bitmask_t ip_flags[] = {
1327        { "DF", IPH_DF, IPH_DF  },
1328        { "MF", IPH_MF, IPH_MF  },
1329        { NULL, 0,      0       }
1330 };

1332 /* ARGSUSED */
1333 static int
1334 iphdr(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1335 {
1336        uint_t          verbose = FALSE, force = FALSE;
1337        ipha_t          iph[1];
1338        uint16_t        ver, totlen, hdrlen, ipid, off, csum;
1339        uintptr_t       nxt_proto;
1340        char            exp_csum[8];

1342        if (mdb_getopts(argc, argv,
```



```
1343            'v', MDB_OPT_SETBITS, TRUE, &verbose,
1344            'f', MDB_OPT_SETBITS, TRUE, &force, NULL) != argc)
1345                return (DCMD_USAGE);

1347        if (mdb_vread(iph, sizeof (*iph), addr) == -1) {
1348                mdb_warn("failed to read IPv4 header at %p", addr);
1349                return (DCMD_ERR);
1350        }

1352        ver = (iph->ipha_version_and_hdr_length & 0xf0) >> 4;
1353        if (ver != IPV4_VERSION) {
1354                if (ver == IPV6_VERSION) {
1355                        return (ip6hdr(addr, flags, argc, argv));
1356                } else if (!force) {
1357                        mdb_warn("unknown IP version: %d\n", ver);
1358                        return (DCMD_ERR);
1359                }
1360        }

1362        mdb_printf("%<b>IPv4 header%</b>\n");
1363        mdb_printf("%-34s %-34s\n"
1364            "%<u>%-4s %-4s %-5s %-5s %-6s %-5s %-5s %-6s %-8s %-6s%</u>\n",
1365            "SRC", "DST",
1366            "HLEN", "TOS", "LEN", "ID", "OFFSET", "TTL", "PROTO", "CHKSUM",
1367            "EXP-CSUM", "FLGS");

1369        hdrlen = (iph->ipha_version_and_hdr_length & 0x0f) << 2;
1370        mdb_nhconvert(&totlen, &iph->ipha_length, sizeof (totlen));
1371        mdb_nhconvert(&ipid, &iph->ipha_ident, sizeof (ipid));
1372        mdb_nhconvert(&off, &iph->ipha_fragment_offset_and_flags, sizeof (off));
1373        if (hdrlen == IP_SIMPLE_HDR_LENGTH) {
1374                if ((csum = ipcksum(iph, sizeof (*iph))) != 0)
1375                        csum = ~(~csum + ~iph->ipha_hdr_checksum);
1376                else
1377                        csum = iph->ipha_hdr_checksum;
1378                mdb_snprintf(exp_csum, 8, "%u", csum);
1379        } else {
1380                mdb_snprintf(exp_csum, 8, "<n/a>");
1381        }

1383        mdb_printf("%-34I %-34I%\n"
1384            "%-4d %-4d %-5hu %-5hu %-6hu %-5hu %-5hu %-6u %-8s <%5hb>\n",
1385            iph->ipha_src, iph->ipha_dst,
1386            hdrlen, iph->ipha_type_of_service, totlen, ipid,
1387            (off << 3) & 0xffff, iph->ipha_ttl, iph->ipha_protocol,
1388            iph->ipha_hdr_checksum, exp_csum, off, ip_flags);

1390        if (verbose) {
1391                nxt_proto = addr + hdrlen;
1392                return (transport_hdr(iph->ipha_protocol, nxt_proto));
1393        } else {
1394                return (DCMD_OK);
1395        }
1396 }

1398 /* ARGSUSED */
1399 static int
1400 ip6hdr(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1401 {
1402        uint_t          verbose = FALSE, force = FALSE;
1403        ip6_t           iph[1];
1404        int             ver, class, flow;
1405        uint16_t        plen;
1406        uintptr_t       nxt_proto;

1408        if (mdb_getopts(argc, argv,
```

```
1409             'v', MDB_OPT_SETBITS, TRUE, &verbose,
1410             'f', MDB_OPT_SETBITS, TRUE, &force, NULL) != argc)
1411                 return (DCMD_USAGE);

1413         if (mdb_vread(iph, sizeof (*iph), addr) == -1) {
1414                 mdb_warn("failed to read IPv6 header at %p", addr);
1415                 return (DCMD_ERR);
1416         }

1418         ver = (iph->ip6_vfc & 0xf0) >> 4;
1419         if (ver != IPV6_VERSION) {
1420                 if (ver == IPV4_VERSION) {
1421                         return (iphdr(addr, flags, argc, argv));
1422                 } else if (!force) {
1423                         mdb_warn("unknown IP version: %d\n", ver);
1424                         return (DCMD_ERR);
1425                 }
1426         }

1428         mdb_printf("%<b>IPv6 header%</b>\n");
1429         mdb_printf("%<u>%-26s %-26s %4s %7s %5s %3s %3s%</u>\n",
1430             "SRC", "DST", "TCLS", "FLOW-ID", "PLEN", "NXT", "HOP");

1432         class = (iph->ip6_vcf & IPV6_FLOWINFO_TCLASS) >> 20;
1433         mdb_nhconvert(&class, &class, sizeof (class));
1434         flow = iph->ip6_vcf & IPV6_FLOWINFO_FLOWLABEL;
1435         mdb_nhconvert(&flow, &flow, sizeof (flow));
1436         mdb_nhconvert(&plen, &iph->ip6_plen, sizeof (plen));

1438         mdb_printf("%-26N %-26N %4d %7d %5hu %3d %3d\n",
1439             &iph->ip6_src, &iph->ip6_dst,
1440             class, flow, plen, iph->ip6_nxt, iph->ip6_hlim);

1442         if (verbose) {
1443                 nxt_proto = addr + sizeof (ip6_t);
1444                 return (transport_hdr(iph->ip6_nxt, nxt_proto));
1445         } else {
1446                 return (DCMD_OK);
1447         }
1448 }

1450 int
1451 nce(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1452 {
1453         nce_t nce;
1454         nce_cbdata_t nce_cb;
1455         int ipversion = 0;
1456         const char *opt_P = NULL, *opt_ill;

1458         if (mdb_getopts(argc, argv,
1459             'i', MDB_OPT_STR, &opt_ill,
1460             'P', MDB_OPT_STR, &opt_P, NULL) != argc)
1461                 return (DCMD_USAGE);

1463         if (opt_P != NULL) {
1464                 if (strcmp("v4", opt_P) == 0) {
1465                         ipversion = IPV4_VERSION;
1466                 } else if (strcmp("v6", opt_P) == 0) {
1467                         ipversion = IPV6_VERSION;
1468                 } else {
1469                         mdb_warn("invalid protocol '%s'\n", opt_P);
1470                         return (DCMD_USAGE);
1471                 }
1472         }

1474         if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {
```



```
1475                 mdb_printf("%<u>%?s %5s %18s %?s %s %s %</u>\n",
1476                     "ADDR", "INTF", "LLADDR", "FP_MP", "REFCNT",
1477                     "NCE_ADDR");
1478         }

1480         bzero(&nce_cb, sizeof (nce_cb));
1481         if (opt_ill != NULL) {
1482                 strcpy(nce_cb.nce_ill_name, opt_ill);
1483         }
1484         nce_cb.nce_ipversion = ipversion;

1486         if (flags & DCMD_ADDRSPEC) {
1487                 (void) mdb_vread(&nce, sizeof (nce_t), addr);
1488                 (void) nce_format(addr, &nce, &nce_cb);
1489         } else if (mdb_walk("nce", (mdb_walk_cb_t)nce_format, &nce_cb) == -1) {
1490                 mdb_warn("failed to walk ire table");
1491                 return (DCMD_ERR);
1492         }

1494         return (DCMD_OK);
1495 }

1497 /* ARGSUSED */
1498 static int
1499 dce_format(uintptr_t addr, const dce_t *dcep, void *dce_cb_arg)
1500 {
1501         static const mdb_bitmask_t dmasks[] = {
1502                 { "D",  DCEF_DEFAULT,           DCEF_DEFAULT },
1503                 { "P",  DCEF_PMTU,              DCEF_PMTU },
1504                 { "U",  DCEF_UINFO,             DCEF_UINFO },
1505                 { "S",  DCEF_TOO_SMALL_PMTU,    DCEF_TOO_SMALL_PMTU },
1506                 { NULL, 0,                      0               }
1507         };
1508         char flagsbuf[2 * A_CNT(dmasks)];
1509         int ipversion = *(int *)dce_cb_arg;
1510         boolean_t condemned = dcep->dce_generation == DCE_GENERATION_CONDEMNED;

1512         if (ipversion != 0 && ipversion != dcep->dce_ipversion)
1513                 return (WALK_NEXT);

1515         mdb_snprintf(flagsbuf, sizeof (flagsbuf), "%b", dcep->dce_flags,
1516             dmasks);

1518         switch (dcep->dce_ipversion) {
1519         case IPV4_VERSION:
1520                 mdb_printf("%<u>%?p%3s %8s %8d %30I %</u>\n", addr, condemned ?
1521                     "(C)" : "", flagsbuf, dcep->dce_pmtu, &dcep->dce_v4addr);
1522                 break;
1523         case IPV6_VERSION:
1524                 mdb_printf("%<u>%?p%3s %8s %8d %30N %</u>\n", addr, condemned ?
1525                     "(C)" : "", flagsbuf, dcep->dce_pmtu, &dcep->dce_v6addr);
1526                 break;
1527         default:
1528                 mdb_printf("%<u>%?p%3s %8s %8d %30s %</u>\n", addr, condemned ?
1529                     "(C)" : "", flagsbuf, dcep->dce_pmtu, "");
1530         }

1532         return (WALK_NEXT);
1533 }

1535 int
1536 dce(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1537 {
1538         dce_t dce;
1539         const char *opt_P = NULL;
1540         const char *zone_name = NULL;
```

```
1541         ip_stack_t *ipst = NULL;
1542         int ipversion = 0;

1544         if (mdb_getopts(argc, argv,
1545             's', MDB_OPT_STR, &zone_name,
1546             'P', MDB_OPT_STR, &opt_P, NULL) != argc)
1547                 return (DCMD_USAGE);

1549         /* Follow the specified zone name to find a ip_stack_t*. */
1550         if (zone_name != NULL) {
1551                 ipst = zone_to_ips(zone_name);
1552                 if (ipst == NULL)
1553                         return (DCMD_USAGE);
1554         }

1556         if (opt_P != NULL) {
1557                 if (strcmp("v4", opt_P) == 0) {
1558                         ipversion = IPV4_VERSION;
1559                 } else if (strcmp("v6", opt_P) == 0) {
1560                         ipversion = IPV6_VERSION;
1561                 } else {
1562                         mdb_warn("invalid protocol '%s'\n", opt_P);
1563                         return (DCMD_USAGE);
1564                 }
1565         }

1567         if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {
1568                 mdb_printf("%<u>%?s%3s %8s %8s %30s %</u>\n",
1569                     "ADDR", "", "FLAGS", "PMTU", "DST_ADDR");
1570         }

1572         if (flags & DCMD_ADDRSPEC) {
1573                 (void) mdb_vread(&dce, sizeof (dce_t), addr);
1574                 (void) dce_format(addr, &dce, &ipversion);
1575         } else if (mdb_pwalk("dce", (mdb_walk_cb_t)dce_format, &ipversion,
1576             (uintptr_t)ipst) == -1) {
1577                 mdb_warn("failed to walk dce cache");
1578                 return (DCMD_ERR);
1579         }

1581         return (DCMD_OK);
1582 }

1584 int
1585 ire(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1586 {
1587         uint_t verbose = FALSE;
1588         ire_t ire;
1589         ire_cbdata_t ire_cb;
1590         int ipversion = 0;
1591         const char *opt_P = NULL;
1592         const char *zone_name = NULL;
1593         ip_stack_t *ipst = NULL;

1595         if (mdb_getopts(argc, argv,
1596             'v', MDB_OPT_SETBITS, TRUE, &verbose,
1597             's', MDB_OPT_STR, &zone_name,
1598             'P', MDB_OPT_STR, &opt_P, NULL) != argc)
1599                 return (DCMD_USAGE);

1601         /* Follow the specified zone name to find a ip_stack_t*. */
1602         if (zone_name != NULL) {
1603                 ipst = zone_to_ips(zone_name);
1604                 if (ipst == NULL)
1605                         return (DCMD_USAGE);
1606         }
```



```
1608         if (opt_P != NULL) {
1609                 if (strcmp("v4", opt_P) == 0) {
1610                         ipversion = IPV4_VERSION;
1611                 } else if (strcmp("v6", opt_P) == 0) {
1612                         ipversion = IPV6_VERSION;
1613                 } else {
1614                         mdb_warn("invalid protocol '%s'\n", opt_P);
1615                         return (DCMD_USAGE);
1616                 }
1617         }

1619         if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {

1621                 if (verbose) {
1622                         mdb_printf("%?s %40s %-20s%\n"
1623                             "%?s %40s %-20s%\n"
1624                             "%<u>%?s %40s %4s %-20s %s%</u>\n",
1625                             "ADDR", "SRC", "TYPE",
1626                             "", "DST", "MARKS",
1627                             "", "STACK", "ZONE", "FLAGS", "INTF");
1628                 } else {
1629                         mdb_printf("%<u>%?s %30s %30s %5s %4s %s%</u>\n",
1630                             "ADDR", "SRC", "DST", "STACK", "ZONE", "INTF");
1631                 }
1632         }

1634         ire_cb.verbose = (verbose == TRUE);
1635         ire_cb.ire_ipversion = ipversion;

1637         if (flags & DCMD_ADDRSPEC) {
1638                 (void) mdb_vread(&ire, sizeof (ire_t), addr);
1639                 (void) ire_format(addr, &ire, &ire_cb);
1640         } else if (mdb_pwalk("ire", (mdb_walk_cb_t)ire_format, &ire_cb,
1641             (uintptr_t)ipst) == -1) {
1642                 mdb_warn("failed to walk ire table");
1643                 return (DCMD_ERR);
1644         }

1646         return (DCMD_OK);
1647 }

1649 static size_t
1650 mi_osize(const queue_t *q)
1651 {
1652         /*
1653          * The code in common/inet/mi.c allocates an extra word to store the
1654          * size of the allocation.  An mi_o_s is thus a size_t plus an mi_o_s.
1655          */
1656         struct mi_block {
1657                 size_t mi_nbytes;
1658                 struct mi_o_s mi_o;
1659         } m;

1661         if (mdb_vread(&m, sizeof (m), (uintptr_t)q->q_ptr -
1662             sizeof (m)) == sizeof (m))
1663                 return (m.mi_nbytes - sizeof (m));

1665         return (0);
1666 }

1668 static void
1669 ip_ill_qinfo(const queue_t *q, char *buf, size_t nbytes)
1670 {
1671         char name[32];
1672         ill_t ill;
```

```
1674         if (mdb_vread(&ill, sizeof (ill),
1675             (uintptr_t)q->q_ptr) == sizeof (ill) &&
1676             mdb_readstr(name, sizeof (name), (uintptr_t)ill.ill_name) > 0)
1677                 (void) mdb_snprintf(buf, nbytes, "if: %s", name);
1678 }

1680 void
1681 ip_qinfo(const queue_t *q, char *buf, size_t nbytes)
1682 {
1683         size_t size = mi_osize(q);

1685         if (size == sizeof (ill_t))
1686                 ip_ill_qinfo(q, buf, nbytes);
1687 }

1689 uintptr_t
1690 ip_rnext(const queue_t *q)
1691 {
1692         size_t size = mi_osize(q);
1693         ill_t ill;

1695         if (size == sizeof (ill_t) && mdb_vread(&ill, sizeof (ill),
1696             (uintptr_t)q->q_ptr) == sizeof (ill))
1697                 return ((uintptr_t)ill.ill_rq);

1699         return (NULL);
1700 }

1702 uintptr_t
1703 ip_wnext(const queue_t *q)
1704 {
1705         size_t size = mi_osize(q);
1706         ill_t ill;

1708         if (size == sizeof (ill_t) && mdb_vread(&ill, sizeof (ill),
1709             (uintptr_t)q->q_ptr) == sizeof (ill))
1710                 return ((uintptr_t)ill.ill_wq);

1712         return (NULL);
1713 }

1715 /*
1716  * Print the core fields in an squeue_t.  With the "-v" argument,
1717  * provide more verbose output.
1718  */
1719 static int
1720 squeue(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1721 {
1722         unsigned int    i;
1723         unsigned int    verbose = FALSE;
1724         const int       SQUEUE_STATEDELT = (int)(sizeof (uintptr_t) + 9);
1725         boolean_t       arm;
1726         squeue_t        squeue;

1728         if (!(flags & DCMD_ADDRSPEC)) {
1729                 if (mdb_walk_dcmd("genunix`squeue_cache", "ip`squeue",
1730                     argc, argv) == -1) {
1731                         mdb_warn("failed to walk squeue cache");
1732                         return (DCMD_ERR);
1733                 }
1734                 return (DCMD_OK);
1735         }

1737         if (mdb_getopts(argc, argv, 'v', MDB_OPT_SETBITS, TRUE, &verbose, NULL)
1738             != argc)
```



```
1739                 return (DCMD_USAGE);

1741         if (!DCMD_HDRSPEC(flags) && verbose)
1742                 mdb_printf("\n\n");

1744         if (DCMD_HDRSPEC(flags) || verbose) {
1745                 mdb_printf("%?s %-5s %-3s %?s %?s %?s\n",
1746                     "ADDR", "STATE", "CPU",
1747                     "FIRST", "LAST", "WORKER");
1748         }

1750         if (mdb_vread(&squeue, sizeof (squeue_t), addr) == -1) {
1751                 mdb_warn("cannot read squeue_t at %p", addr);
1752                 return (DCMD_ERR);
1753         }

1755         mdb_printf("%0?p %05x %3d %0?p %0?p %0?p\n",
1756             addr, squeue.sq_state, squeue.sq_bind,
1757             squeue.sq_first, squeue.sq_last, squeue.sq_worker);

1759         if (!verbose)
1760                 return (DCMD_OK);

1762         arm = B_TRUE;
1763         for (i = 0; squeue_states[i].bit_name != NULL; i++) {
1764                 if (((squeue.sq_state) & (1 << i)) == 0)
1765                         continue;

1767                 if (arm) {
1768                         mdb_printf("%*s|\n", SQUEUE_STATEDELT, "");
1769                         mdb_printf("%*s+-->  ", SQUEUE_STATEDELT, "");
1770                         arm = B_FALSE;
1771                 } else
1772                         mdb_printf("%*s      ", SQUEUE_STATEDELT, "");

1774                 mdb_printf("%-12s %s\n", squeue_states[i].bit_name,
1775                     squeue_states[i].bit_descr);
1776         }

1778         return (DCMD_OK);
1779 }

1781 static void
1782 ip_squeue_help(void)
1783 {
1784         mdb_printf("Print the core information for a given NCA squeue_t.\n\n");
1785         mdb_printf("Options:\n");
1786         mdb_printf("\t-v\tbe verbose (more descriptive)\n");
1787 }

1789 /*
1790  * This is called by ::th_trace (via a callback) when walking the th_hash
1791  * list.  It calls modent to find the entries.
1792  */
1793 /* ARGSUSED */
1794 static int
1795 modent_summary(uintptr_t addr, const void *data, void *private)
1796 {
1797         th_walk_data_t *thw = private;
1798         const struct mod_hash_entry *mhe = data;
1799         th_trace_t th;

1801         if (mdb_vread(&th, sizeof (th), (uintptr_t)mhe->mhe_val) == -1) {
1802                 mdb_warn("failed to read th_trace_t %p", mhe->mhe_val);
1803                 return (WALK_ERR);
1804         }
```

```
1806         if (th.th_refcnt == 0 && thw->thw_non_zero_only)
1807                 return (WALK_NEXT);

1809         if (!thw->thw_match) {
1810                 mdb_printf("%?p %?p %?p %8d %?p\n", thw->thw_ipst, mhe->mhe_key,
1811                     mhe->mhe_val, th.th_refcnt, th.th_id);
1812         } else if (thw->thw_matchkey == (uintptr_t)mhe->mhe_key) {
1813                 int i, j, k;
1814                 tr_buf_t *tr;

1816                 mdb_printf("Object %p in IP stack %p:\n", mhe->mhe_key,
1817                     thw->thw_ipst);
1818                 i = th.th_trace_lastref;
1819                 mdb_printf("\tThread %p refcnt %d:\n", th.th_id,
1820                     th.th_refcnt);
1821                 for (j = TR_BUF_MAX; j > 0; j--) {
1822                         tr = th.th_trbuf + i;
1823                         if (tr->tr_depth == 0 || tr->tr_depth > TR_STACK_DEPTH)
1824                                 break;
1825                         mdb_printf("\t  T%+ld:\n", tr->tr_time -
1826                             thw->thw_lbolt);
1827                         for (k = 0; k < tr->tr_depth; k++)
1828                                 mdb_printf("\t\t%a\n", tr->tr_stack[k]);
1829                         if (--i < 0)
1830                                 i = TR_BUF_MAX - 1;
1831                 }
1832         }
1833         return (WALK_NEXT);
1834 }

1836 /*
1837  * This is called by ::th_trace (via a callback) when walking the th_hash
1838  * list.  It calls modent to find the entries.
1839  */
1840 /* ARGSUSED */
1841 static int
1842 th_hash_summary(uintptr_t addr, const void *data, void *private)
1843 {
1844         const th_hash_t *thh = data;
1845         th_walk_data_t *thw = private;

1847         thw->thw_ipst = (uintptr_t)thh->thh_ipst;
1848         return (mdb_pwalk("modent", modent_summary, private,
1849             (uintptr_t)thh->thh_hash));
1850 }

1852 /*
1853  * Print or summarize the th_trace_t structures.
1854  */
1855 static int
1856 th_trace(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1857 {
1858         th_walk_data_t thw;

1860         (void) memset(&thw, 0, sizeof (thw));

1862         if (mdb_getopts(argc, argv,
1863             'n', MDB_OPT_SETBITS, TRUE, &thw.thw_non_zero_only,
1864             NULL) != argc)
1865                 return (DCMD_USAGE);

1867         if (!(flags & DCMD_ADDRSPEC)) {
1868                 /*
1869                  * No address specified.  Walk all of the th_hash_t in the
1870                  * system, and summarize the th_trace_t entries in each.
```



```
1871                  */
1872                 mdb_printf("%?s %?s %?s %8s %?s\n",
1873                     "IPSTACK", "OBJECT", "TRACE", "REFCNT", "THREAD");
1874                 thw.thw_match = B_FALSE;
1875         } else {
1876                 thw.thw_match = B_TRUE;
1877                 thw.thw_matchkey = addr;

1879                 if ((thw.thw_lbolt = (clock_t)mdb_get_lbolt()) == -1) {
1880                         mdb_warn("failed to read lbolt");
1881                         return (DCMD_ERR);
1882                 }
1883         }
1884         if (mdb_pwalk("th_hash", th_hash_summary, &thw, NULL) == -1) {
1885                 mdb_warn("can't walk th_hash entries");
1886                 return (DCMD_ERR);
1887         }
1888         return (DCMD_OK);
1889 }

1891 static void
1892 th_trace_help(void)
1893 {
1894         mdb_printf("If given an address of an ill_t, ipif_t, ire_t, or ncec_t, "
1895             "print the\n"
1896             "corresponding th_trace_t structure in detail.  Otherwise, if no "
1897             "address is\n"
1898             "given, then summarize all th_trace_t structures.\n\n");
1899         mdb_printf("Options:\n"
1900             "\t-n\tdisplay only entries with non-zero th_refcnt\n");
1901 }

1903 static const mdb_dcmd_t dcmds[] = {
1904         { "conn_status", ":",
1905             "display connection structures from ipcl hash tables",
1906             conn_status, conn_status_help },
1907         { "srcid_status", ":",
1908             "display connection structures from ipcl hash tables",
1909             srcid_status },
1910         { "ill", "?[-v] [-P v4 | v6] [-s exclusive-ip-zone-name]",
1911             "display ill_t structures", ill, ill_help },
1912         { "illif", "?[-P v4 | v6]",
1913             "display or filter IP Lower Level InterFace structures", illif,
1914             illif_help },
1915         { "iphdr", ":[-vf]", "display an IPv4 header", iphdr },
1916         { "ip6hdr", ":[-vf]", "display an IPv6 header", ip6hdr },
1917         { "ipif", "?[-v] [-P v4 | v6]", "display ipif structures",
1918             ipif, ipif_help },
1919         { "ire", "?[-v] [-P v4|v6] [-s exclusive-ip-zone-name]",
1920             "display Internet Route Entry structures", ire },
1921         { "nce", "?[-P v4|v6] [-i <interface>]",
1922             "display interface-specific Neighbor Cache structures", nce },
1923         { "ncec", "?[-P v4 | v6]", "display Neighbor Cache Entry structures",
1924             ncec },
1925         { "dce", "?[-P v4|v6] [-s exclusive-ip-zone-name]",
1926             "display Destination Cache Entry structures", dce },
1927         { "squeue", ":[-v]", "print core squeue_t info", squeue,
1928             ip_squeue_help },
1929         { "tcphdr", ":", "display a TCP header", tcphdr },
1930         { "udphdr", ":", "display an UDP header", udphdr },
1931         { "sctphdr", ":", "display an SCTP header", sctphdr },
1932         { "th_trace", "?[-n]", "display th_trace_t structures", th_trace,
1933             th_trace_help },
1934         { NULL }
1935 };
```

```
1937 static const mdb_walker_t walkers[] = {
1938         { "conn_status", "walk list of conn_t structures",
1939                 ip_stacks_common_walk_init, conn_status_walk_step, NULL },
1940         { "illif", "walk list of ill interface types for all stacks",
1941                 ip_stacks_common_walk_init, illif_walk_step, NULL },
1942         { "illif_stack", "walk list of ill interface types",
1943                 illif_stack_walk_init, illif_stack_walk_step,
1944                 illif_stack_walk_fini },
1945         { "ill", "walk active ill_t structures for all stacks",
1946                 ill_walk_init, ill_walk_step, NULL },
1947         { "ipif", "walk list of ipif structures for all stacks",
1948                 ipif_walk_init, ipif_walk_step, NULL },
1949         { "ipif_list", "walk the linked list of ipif structures "
1950                 "for a given ill",
1951                 ip_list_walk_init, ip_list_walk_step,
1952                 ip_list_walk_fini, &ipif_walk_arg },
1953         { "srcid", "walk list of srcid_map structures for all stacks",
1954                 ip_stacks_common_walk_init, srcid_walk_step, NULL },
1955         { "srcid_list", "walk list of srcid_map structures for a stack",
1956                 ip_list_walk_init, ip_list_walk_step, ip_list_walk_fini,
1957                 &srcid_walk_arg },
1958         { "ire", "walk active ire_t structures",
1959                 ire_walk_init, ire_walk_step, NULL },
1960         { "ire_next", "walk ire_t structures in the ctable",
1961                 ire_next_walk_init, ire_next_walk_step, NULL },
1962         { "nce", "walk active nce_t structures",
1963                 nce_walk_init, nce_walk_step, NULL },
1964         { "dce", "walk active dce_t structures",
1965                 dce_walk_init, dce_walk_step, NULL },
1966         { "dccp_stacks", "walk all the dccp_stack_t",
1967                 ns_walk_init, dccp_stacks_walk_step, NULL },
1968 #endif /* ! codereview */
1969         { "ip_stacks", "walk all the ip_stack_t",
1970                 ns_walk_init, ip_stacks_walk_step, NULL },
1971         { "tcp_stacks", "walk all the tcp_stack_t",
1972                 ns_walk_init, tcp_stacks_walk_step, NULL },
1973         { "sctp_stacks", "walk all the sctp_stack_t",
1974                 ns_walk_init, sctp_stacks_walk_step, NULL },
1975         { "udp_stacks", "walk all the udp_stack_t",
1976                 ns_walk_init, udp_stacks_walk_step, NULL },
1977         { "th_hash", "walk all the th_hash_t entries",
1978                 th_hash_walk_init, th_hash_walk_step, NULL },
1979         { "ncec", "walk list of ncec structures for all stacks",
1980                 ip_stacks_common_walk_init, ncec_walk_step, NULL },
1981         { "ncec_stack", "walk list of ncec structures",
1982                 ncec_stack_walk_init, ncec_stack_walk_step,
1983                 ncec_stack_walk_fini},
1984         { "udp_hash", "walk list of conn_t structures in ips_ipcl_udp_fanout",
1985                 ipcl_hash_walk_init, ipcl_hash_walk_step,
1986                 ipcl_hash_walk_fini, &udp_hash_arg},
1987         { "conn_hash", "walk list of conn_t structures in ips_ipcl_conn_fanout",
1988                 ipcl_hash_walk_init, ipcl_hash_walk_step,
1989                 ipcl_hash_walk_fini, &conn_hash_arg},
1990         { "bind_hash", "walk list of conn_t structures in ips_ipcl_bind_fanout",
1991                 ipcl_hash_walk_init, ipcl_hash_walk_step,
1992                 ipcl_hash_walk_fini, &bind_hash_arg},
1993         { "proto_hash", "walk list of conn_t structures in "
1994             "ips_ipcl_proto_fanout",
1995                 ipcl_hash_walk_init, ipcl_hash_walk_step,
1996                 ipcl_hash_walk_fini, &proto_hash_arg},
1997         { "proto_v6_hash", "walk list of conn_t structures in "
1998             "ips_ipcl_proto_fanout_v6",
1999                 ipcl_hash_walk_init, ipcl_hash_walk_step,
2000                 ipcl_hash_walk_fini, &proto_v6_hash_arg},
2001         { "ilb_stacks", "walk all ilb_stack_t",
2002                 ns_walk_init, ilb_stacks_walk_step, NULL },
```



```
2003         { "ilb_rules", "walk ilb rules in a given ilb_stack_t",
2004                 ilb_rules_walk_init, ilb_rules_walk_step, NULL },
2005         { "ilb_servers", "walk server in a given ilb_rule_t",
2006                 ilb_servers_walk_init, ilb_servers_walk_step, NULL },
2007         { "ilb_nat_src", "walk NAT source table of a given ilb_stack_t",
2008                 ilb_nat_src_walk_init, ilb_nat_src_walk_step,
2009                 ilb_common_walk_fini },
2010         { "ilb_conns", "walk NAT table of a given ilb_stack_t",
2011                 ilb_conn_walk_init, ilb_conn_walk_step, ilb_common_walk_fini },
2012         { "ilb_stickys", "walk sticky table of a given ilb_stack_t",
2013                 ilb_sticky_walk_init, ilb_sticky_walk_step,
2014                 ilb_common_walk_fini },
2015         { "tcps_sc", "walk all the per CPU stats counters of a tcp_stack_t",
2016                 tcps_sc_walk_init, tcps_sc_walk_step, NULL },
2017         { NULL }
2018 };

2020 static const mdb_qops_t ip_qops = { ip_qinfo, ip_rnext, ip_wnext };
2021 static const mdb_modinfo_t modinfo = { MDB_API_VERSION, dcmds, walkers };

2023 const mdb_modinfo_t *
2024 _mdb_init(void)
2025 {
2026         GElf_Sym sym;

2028         if (mdb_lookup_by_obj("ip", "ipwinit", &sym) == 0)
2029                 mdb_qops_install(&ip_qops, (uintptr_t)sym.st_value);

2031         return (&modinfo);
2032 }

2034 void
2035 _mdb_fini(void)
2036 {
2037         GElf_Sym sym;

2039         if (mdb_lookup_by_obj("ip", "ipwinit", &sym) == 0)
2040                 mdb_qops_remove(&ip_qops, (uintptr_t)sym.st_value);
2041 }

2043 static char *
2044 ncec_state(int ncec_state)
2045 {
2046         switch (ncec_state) {
2047         case ND_UNCHANGED:
2048                 return ("unchanged");
2049         case ND_INCOMPLETE:
2050                 return ("incomplete");
2051         case ND_REACHABLE:
2052                 return ("reachable");
2053         case ND_STALE:
2054                 return ("stale");
2055         case ND_DELAY:
2056                 return ("delay");
2057         case ND_PROBE:
2058                 return ("probe");
2059         case ND_UNREACHABLE:
2060                 return ("unreach");
2061         case ND_INITIAL:
2062                 return ("initial");
2063         default:
2064                 return ("??");
2065         }
2066 }

2068 static char *
```

```
2069 ncec_l2_addr(const ncec_t *ncec, const ill_t *ill)
2070 {
2071         uchar_t *h;
2072         static char addr_buf[L2MAXADDRSTRLEN];

2074         if (ncec->ncec_lladdr == NULL) {
2075                 return ("None");
2076         }

2078         if (ill->ill_net_type == IRE_IF_RESOLVER) {

2080                 if (ill->ill_phys_addr_length == 0)
2081                         return ("None");
2082                 h = mdb_zalloc(ill->ill_phys_addr_length, UM_SLEEP);
2083                 if (mdb_vread(h, ill->ill_phys_addr_length,
2084                     (uintptr_t)ncec->ncec_lladdr) == -1) {
2085                         mdb_warn("failed to read hwaddr at %p",
2086                             ncec->ncec_lladdr);
2087                         return ("Unknown");
2088                 }
2089                 mdb_mac_addr(h, ill->ill_phys_addr_length,
2090                     addr_buf, sizeof (addr_buf));
2091         } else {
2092                 return ("None");
2093         }
2094         mdb_free(h, ill->ill_phys_addr_length);
2095         return (addr_buf);
2096 }

2098 static char *
2099 nce_l2_addr(const nce_t *nce, const ill_t *ill)
2100 {
2101         uchar_t *h;
2102         static char addr_buf[L2MAXADDRSTRLEN];
2103         mblk_t mp;
2104         size_t mblen;

2106         if (nce->nce_dlur_mp == NULL)
2107                 return ("None");

2109         if (ill->ill_net_type == IRE_IF_RESOLVER) {
2110                 if (mdb_vread(&mp, sizeof (mblk_t),
2111                     (uintptr_t)nce->nce_dlur_mp) == -1) {
2112                         mdb_warn("failed to read nce_dlur_mp at %p",
2113                             nce->nce_dlur_mp);
2114                         return ("None");
2115                 }
2116                 if (ill->ill_phys_addr_length == 0)
2117                         return ("None");
2118                 mblen = mp.b_wptr - mp.b_rptr;
2119                 if (mblen > (sizeof (dl_unitdata_req_t) + MAX_SAP_LEN) ||
2120                     ill->ill_phys_addr_length > MAX_SAP_LEN ||
2121                     (NCE_LL_ADDR_OFFSET(ill) +
2122                     ill->ill_phys_addr_length) > mblen) {
2123                         return ("Unknown");
2124                 }
2125                 h = mdb_zalloc(mblen, UM_SLEEP);
2126                 if (mdb_vread(h, mblen, (uintptr_t)(mp.b_rptr)) == -1) {
2127                         mdb_warn("failed to read hwaddr at %p",
2128                             mp.b_rptr + NCE_LL_ADDR_OFFSET(ill));
2129                         return ("Unknown");
2130                 }
2131                 mdb_mac_addr(h + NCE_LL_ADDR_OFFSET(ill),
2132                     ill->ill_phys_addr_length, addr_buf, sizeof (addr_buf));
2133         } else {
2134                 return ("None");
```



```
2135         }
2136         mdb_free(h, mblen);
2137         return (addr_buf);
2138 }

2140 static void
2141 ncec_header(uint_t flags)
2142 {
2143         if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {

2145                 mdb_printf("%<u>%?s %-20s %-10s %-8s %-5s %s%</u>\n",
2146                     "ADDR", "HW_ADDR", "STATE", "FLAGS", "ILL", "IP ADDR");
2147         }
2148 }

2150 int
2151 ncec(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
2152 {
2153         ncec_t ncec;
2154         ncec_cbdata_t id;
2155         int ipversion = 0;
2156         const char *opt_P = NULL;

2158         if (mdb_getopts(argc, argv,
2159             'P', MDB_OPT_STR, &opt_P, NULL) != argc)
2160                 return (DCMD_USAGE);

2162         if (opt_P != NULL) {
2163                 if (strcmp("v4", opt_P) == 0) {
2164                         ipversion = IPV4_VERSION;
2165                 } else if (strcmp("v6", opt_P) == 0) {
2166                         ipversion = IPV6_VERSION;
2167                 } else {
2168                         mdb_warn("invalid protocol '%s'\n", opt_P);
2169                         return (DCMD_USAGE);
2170                 }
2171         }

2173         if (flags & DCMD_ADDRSPEC) {

2175                 if (mdb_vread(&ncec, sizeof (ncec_t), addr) == -1) {
2176                         mdb_warn("failed to read ncec at %p\n", addr);
2177                         return (DCMD_ERR);
2178                 }
2179                 if (ipversion != 0 && ncec.ncec_ipversion != ipversion) {
2180                         mdb_printf("IP Version mismatch\n");
2181                         return (DCMD_ERR);
2182                 }
2183                 ncec_header(flags);
2184                 return (ncec_format(addr, &ncec, ipversion));

2186         } else {
2187                 id.ncec_addr = addr;
2188                 id.ncec_ipversion = ipversion;
2189                 ncec_header(flags);
2190                 if (mdb_walk("ncec", (mdb_walk_cb_t)ncec_cb, &id) == -1) {
2191                         mdb_warn("failed to walk ncec table\n");
2192                         return (DCMD_ERR);
2193                 }
2194         }
2195         return (DCMD_OK);
2196 }

2198 static int
2199 ncec_format(uintptr_t addr, const ncec_t *ncec, int ipversion)
2200 {
```

```
2201         static const mdb_bitmask_t ncec_flags[] = {
2202                 { "P",  NCE_F_NONUD,            NCE_F_NONUD },
2203                 { "R",  NCE_F_ISROUTER,         NCE_F_ISROUTER  },
2204                 { "N",  NCE_F_NONUD,            NCE_F_NONUD     },
2205                 { "A",  NCE_F_ANYCAST,          NCE_F_ANYCAST   },
2206                 { "C",  NCE_F_CONDEMNED,        NCE_F_CONDEMNED },
2207                 { "U",  NCE_F_UNSOL_ADV,        NCE_F_UNSOL_ADV },
2208                 { "B",  NCE_F_BCAST,            NCE_F_BCAST     },
2209                 { NULL, 0,                      0               }
2210         };
2211 #define NCE_MAX_FLAGS   (sizeof (ncec_flags) / sizeof (mdb_bitmask_t))
2212         struct in_addr nceaddr;
2213         ill_t ill;
2214         char ill_name[LIFNAMSIZ];
2215         char flagsbuf[NCE_MAX_FLAGS];

2217         if (mdb_vread(&ill, sizeof (ill), (uintptr_t)ncec->ncec_ill) == -1) {
2218                 mdb_warn("failed to read ncec_ill at %p",
2219                     ncec->ncec_ill);
2220                 return (DCMD_ERR);
2221         }

2223         (void) mdb_readstr(ill_name, MIN(LIFNAMSIZ, ill.ill_name_length),
2224             (uintptr_t)ill.ill_name);

2226         mdb_snprintf(flagsbuf, sizeof (flagsbuf), "%hb",
2227             ncec->ncec_flags, ncec_flags);

2229         if (ipversion != 0 && ncec->ncec_ipversion != ipversion)
2230                 return (DCMD_OK);

2232         if (ncec->ncec_ipversion == IPV4_VERSION) {
2233                 IN6_V4MAPPED_TO_INADDR(&ncec->ncec_addr, &nceaddr);
2234                 mdb_printf("%?p %-20s %-10s "
2235                     "%-8s "
2236                     "%-5s %I\n",
2237                     addr, ncec_l2_addr(ncec, &ill),
2238                     ncec_state(ncec->ncec_state),
2239                     flagsbuf,
2240                     ill_name, nceaddr.s_addr);
2241         } else {
2242                 mdb_printf("%?p %-20s %-10s %-8s %-5s %N\n",
2243                     addr,  ncec_l2_addr(ncec, &ill),
2244                     ncec_state(ncec->ncec_state),
2245                     flagsbuf,
2246                     ill_name, &ncec->ncec_addr);
2247         }

2249         return (DCMD_OK);
2250 }

2252 static uintptr_t
2253 ncec_get_next_hash_tbl(uintptr_t start, int *index, struct ndp_g_s ndp)
2254 {
2255         uintptr_t addr = start;
2256         int i = *index;

2258         while (addr == NULL) {

2260                 if (++i >= NCE_TABLE_SIZE)
2261                         break;
2262                 addr = (uintptr_t)ndp.nce_hash_tbl[i];
2263         }
2264         *index = i;
2265         return (addr);
2266 }
```



```
2268 static int
2269 ncec_walk_step(mdb_walk_state_t *wsp)
2270 {
2271         uintptr_t kaddr4, kaddr6;

2273         kaddr4 = wsp->walk_addr + OFFSETOF(ip_stack_t, ips_ndp4);
2274         kaddr6 = wsp->walk_addr + OFFSETOF(ip_stack_t, ips_ndp6);

2276         if (mdb_vread(&kaddr4, sizeof (kaddr4), kaddr4) == -1) {
2277                 mdb_warn("can't read ips_ip_cache_table at %p", kaddr4);
2278                 return (WALK_ERR);
2279         }
2280         if (mdb_vread(&kaddr6, sizeof (kaddr6), kaddr6) == -1) {
2281                 mdb_warn("can't read ips_ip_cache_table at %p", kaddr6);
2282                 return (WALK_ERR);
2283         }
2284         if (mdb_pwalk("ncec_stack", wsp->walk_callback, wsp->walk_cbdata,
2285             kaddr4) == -1) {
2286                 mdb_warn("couldn't walk 'ncec_stack' for ips_ndp4 %p",
2287                     kaddr4);
2288                 return (WALK_ERR);
2289         }
2290         if (mdb_pwalk("ncec_stack", wsp->walk_callback,
2291             wsp->walk_cbdata, kaddr6) == -1) {
2292                 mdb_warn("couldn't walk 'ncec_stack' for ips_ndp6 %p",
2293                     kaddr6);
2294                 return (WALK_ERR);
2295         }
2296         return (WALK_NEXT);
2297 }

2299 static uintptr_t
2300 ipcl_hash_get_next_connf_tbl(ipcl_hash_walk_data_t *iw)
2301 {
2302         struct connf_s connf;
2303         uintptr_t addr = NULL, next;
2304         int index = iw->connf_tbl_index;

2306         do {
2307                 next = iw->hash_tbl + index * sizeof (struct connf_s);
2308                 if (++index >= iw->hash_tbl_size) {
2309                         addr = NULL;
2310                         break;
2311                 }
2312                 if (mdb_vread(&connf, sizeof (struct connf_s), next) == -1)  {
2313                         mdb_warn("failed to read conn_t at %p", next);
2314                         return (NULL);
2315                 }
2316                 addr = (uintptr_t)connf.connf_head;
2317         } while (addr == NULL);
2318         iw->connf_tbl_index = index;
2319         return (addr);
2320 }

2322 static int
2323 ipcl_hash_walk_init(mdb_walk_state_t *wsp)
2324 {
2325         const hash_walk_arg_t *arg = wsp->walk_arg;
2326         ipcl_hash_walk_data_t *iw;
2327         uintptr_t tbladdr;
2328         uintptr_t sizeaddr;

2330         iw = mdb_alloc(sizeof (ipcl_hash_walk_data_t), UM_SLEEP);
2331         iw->conn = mdb_alloc(sizeof (conn_t), UM_SLEEP);
2332         tbladdr = wsp->walk_addr + arg->tbl_off;
```

```
2333         sizeaddr = wsp->walk_addr + arg->size_off;

2335         if (mdb_vread(&iw->hash_tbl, sizeof (uintptr_t), tbladdr) == -1) {
2336                 mdb_warn("can't read fanout table addr at %p", tbladdr);
2337                 mdb_free(iw->conn, sizeof (conn_t));
2338                 mdb_free(iw, sizeof (ipcl_hash_walk_data_t));
2339                 return (WALK_ERR);
2340         }
2341         if (arg->tbl_off == OFFSETOF(ip_stack_t, ips_ipcl_proto_fanout_v4) ||
2342             arg->tbl_off == OFFSETOF(ip_stack_t, ips_ipcl_proto_fanout_v6)) {
2343                 iw->hash_tbl_size = IPPROTO_MAX;
2344         } else {
2345                 if (mdb_vread(&iw->hash_tbl_size, sizeof (int),
2346                     sizeaddr) == -1) {
2347                         mdb_warn("can't read fanout table size addr at %p",
2348                             sizeaddr);
2349                         mdb_free(iw->conn, sizeof (conn_t));
2350                         mdb_free(iw, sizeof (ipcl_hash_walk_data_t));
2351                         return (WALK_ERR);
2352                 }
2353         }
2354         iw->connf_tbl_index = 0;
2355         wsp->walk_addr = ipcl_hash_get_next_connf_tbl(iw);
2356         wsp->walk_data = iw;

2358         if (wsp->walk_addr != NULL)
2359                 return (WALK_NEXT);
2360         else
2361                 return (WALK_DONE);
2362 }

2364 static int
2365 ipcl_hash_walk_step(mdb_walk_state_t *wsp)
2366 {
2367         uintptr_t addr = wsp->walk_addr;
2368         ipcl_hash_walk_data_t *iw = wsp->walk_data;
2369         conn_t *conn = iw->conn;
2370         int ret = WALK_DONE;

2372         while (addr != NULL) {
2373                 if (mdb_vread(conn, sizeof (conn_t), addr) == -1) {
2374                         mdb_warn("failed to read conn_t at %p", addr);
2375                         return (WALK_ERR);
2376                 }
2377                 ret = wsp->walk_callback(addr, iw, wsp->walk_cbdata);
2378                 if (ret != WALK_NEXT)
2379                         break;
2380                 addr = (uintptr_t)conn->conn_next;
2381         }
2382         if (ret == WALK_NEXT) {
2383                 wsp->walk_addr = ipcl_hash_get_next_connf_tbl(iw);

2385                 if (wsp->walk_addr != NULL)
2386                         return (WALK_NEXT);
2387                 else
2388                         return (WALK_DONE);
2389         }

2391         return (ret);
2392 }

2394 static void
2395 ipcl_hash_walk_fini(mdb_walk_state_t *wsp)
2396 {
2397         ipcl_hash_walk_data_t *iw = wsp->walk_data;
```



```
2399         mdb_free(iw->conn, sizeof (conn_t));
2400         mdb_free(iw, sizeof (ipcl_hash_walk_data_t));
2401 }

2403 /*
2404  * Called with walk_addr being the address of ips_ndp{4,6}
2405  */
2406 static int
2407 ncec_stack_walk_init(mdb_walk_state_t *wsp)
2408 {
2409         ncec_walk_data_t *nw;

2411         if (wsp->walk_addr == NULL) {
2412                 mdb_warn("ncec_stack requires ndp_g_s address\n");
2413                 return (WALK_ERR);
2414         }

2416         nw = mdb_alloc(sizeof (ncec_walk_data_t), UM_SLEEP);

2418         if (mdb_vread(&nw->ncec_ip_ndp, sizeof (struct ndp_g_s),
2419             wsp->walk_addr) == -1) {
2420                 mdb_warn("failed to read 'ip_ndp' at %p",
2421                     wsp->walk_addr);
2422                 mdb_free(nw, sizeof (ncec_walk_data_t));
2423                 return (WALK_ERR);
2424         }

2426         /*
2427          * ncec_get_next_hash_tbl() starts at ++i , so initialize index to -1
2428          */
2429         nw->ncec_hash_tbl_index = -1;
2430         wsp->walk_addr = ncec_get_next_hash_tbl(NULL,
2431             &nw->ncec_hash_tbl_index, nw->ncec_ip_ndp);
2432         wsp->walk_data = nw;

2434         return (WALK_NEXT);
2435 }

2437 static int
2438 ncec_stack_walk_step(mdb_walk_state_t *wsp)
2439 {
2440         uintptr_t addr = wsp->walk_addr;
2441         ncec_walk_data_t *nw = wsp->walk_data;

2443         if (addr == NULL)
2444                 return (WALK_DONE);

2446         if (mdb_vread(&nw->ncec, sizeof (ncec_t), addr) == -1) {
2447                 mdb_warn("failed to read ncec_t at %p", addr);
2448                 return (WALK_ERR);
2449         }

2451         wsp->walk_addr = (uintptr_t)nw->ncec.ncec_next;

2453         wsp->walk_addr = ncec_get_next_hash_tbl(wsp->walk_addr,
2454             &nw->ncec_hash_tbl_index, nw->ncec_ip_ndp);

2456         return (wsp->walk_callback(addr, nw, wsp->walk_cbdata));
2457 }

2459 static void
2460 ncec_stack_walk_fini(mdb_walk_state_t *wsp)
2461 {
2462         mdb_free(wsp->walk_data, sizeof (ncec_walk_data_t));
2463 }
```

```
/* ARGSUSED */
static int
ncec_cb(uintptr_t addr, const ncec_walk_data_t *iw, ncec_cbdata_t *id)
{
        ncec_t ncec;

        if (mdb_vread(&ncec, sizeof (ncec_t), addr) == -1) {
                mdb_warn("failed to read ncec at %p", addr);
                return (WALK_NEXT);
        }
        (void) ncec_format(addr, &ncec, id->ncec_ipversion);
        return (WALK_NEXT);
}

static int
ill_walk_init(mdb_walk_state_t *wsp)
{
        if (mdb_layered_walk("illif", wsp) == -1) {
                mdb_warn("can't walk 'illif'");
                return (WALK_ERR);
        }
        return (WALK_NEXT);
}

static int
ill_walk_step(mdb_walk_state_t *wsp)
{
        ill_if_t ill_if;

        if (mdb_vread(&ill_if, sizeof (ill_if_t), wsp->walk_addr) == -1) {
                mdb_warn("can't read ill_if_t at %p", wsp->walk_addr);
                return (WALK_ERR);
        }
        wsp->walk_addr = (uintptr_t)(wsp->walk_addr +
            offsetof(ill_if_t, illif_avl_by_ppa));
        if (mdb_pwalk("avl", wsp->walk_callback, wsp->walk_cbdata,
            wsp->walk_addr) == -1) {
                mdb_warn("can't walk 'avl'");
                return (WALK_ERR);
        }

        return (WALK_NEXT);
}

/* ARGSUSED */
static int
ill_cb(uintptr_t addr, const ill_walk_data_t *iw, ill_cbdata_t *id)
{
        ill_t ill;

        if (mdb_vread(&ill, sizeof (ill_t), (uintptr_t)addr) == -1) {
                mdb_warn("failed to read ill at %p", addr);
                return (WALK_NEXT);
        }

        /* If ip_stack_t is specified, skip ILLs that don't belong to it. */
        if (id->ill_ipst != NULL && ill.ill_ipst != id->ill_ipst)
                return (WALK_NEXT);

        return (ill_format((uintptr_t)addr, &ill, id));
}

static void
ill_header(boolean_t verbose)
{
        if (verbose) {
```



```
                mdb_printf("%-?s %-8s %3s %-10s %-?s %-?s %-10s%</u>\n",
                    "ADDR", "NAME", "VER", "TYPE", "WQ", "IPST", "FLAGS");
                mdb_printf("%-?s %4s%4s %-?s\n",
                    "PHYINT", "CNT", "", "GROUP");
                mdb_printf("%<u>%80s%</u>\n", "");
        } else {
                mdb_printf("%<u>%-?s %-8s %-3s %-10s %4s %-?s %-10s%</u>\n",
                    "ADDR", "NAME", "VER", "TYPE", "CNT", "WQ", "FLAGS");
        }
}

static int
ill_format(uintptr_t addr, const void *illptr, void *ill_cb_arg)
{
        ill_t *ill = (ill_t *)illptr;
        ill_cbdata_t *illcb = ill_cb_arg;
        boolean_t verbose = illcb->verbose;
        phyint_t phyi;
        static const mdb_bitmask_t fmasks[] = {
                { "R",          PHYI_RUNNING,           PHYI_RUNNING    },
                { "P",          PHYI_PROMISC,           PHYI_PROMISC    },
                { "V",          PHYI_VIRTUAL,           PHYI_VIRTUAL    },
                { "I",          PHYI_IPMP,              PHYI_IPMP       },
                { "f",          PHYI_FAILED,            PHYI_FAILED     },
                { "S",          PHYI_STANDBY,           PHYI_STANDBY    },
                { "i",          PHYI_INACTIVE,          PHYI_INACTIVE   },
                { "O",          PHYI_OFFLINE,           PHYI_OFFLINE    },
                { "T",          ILLF_NOTRAILERS,        ILLF_NOTRAILERS },
                { "A",          ILLF_NOARP,             ILLF_NOARP      },
                { "M",          ILLF_MULTICAST,         ILLF_MULTICAST  },
                { "F",          ILLF_ROUTER,            ILLF_ROUTER     },
                { "D",          ILLF_NONUD,             ILLF_NONUD      },
                { "X",          ILLF_NORTEXCH,          ILLF_NORTEXCH   },
                { NULL,         0,                      0               }
        };
        static const mdb_bitmask_t v_fmasks[] = {
                { "RUNNING",    PHYI_RUNNING,           PHYI_RUNNING    },
                { "PROMISC",    PHYI_PROMISC,           PHYI_PROMISC    },
                { "VIRTUAL",    PHYI_VIRTUAL,           PHYI_VIRTUAL    },
                { "IPMP",       PHYI_IPMP,              PHYI_IPMP       },
                { "FAILED",     PHYI_FAILED,            PHYI_FAILED     },
                { "STANDBY",    PHYI_STANDBY,           PHYI_STANDBY    },
                { "INACTIVE",   PHYI_INACTIVE,          PHYI_INACTIVE   },
                { "OFFLINE",    PHYI_OFFLINE,           PHYI_OFFLINE    },
                { "NOTRAILER",  ILLF_NOTRAILERS,        ILLF_NOTRAILERS },
                { "NOARP",      ILLF_NOARP,             ILLF_NOARP      },
                { "MULTICAST",  ILLF_MULTICAST,         ILLF_MULTICAST  },
                { "ROUTER",     ILLF_ROUTER,            ILLF_ROUTER     },
                { "NONUD",      ILLF_NONUD,             ILLF_NONUD      },
                { "NORTEXCH",   ILLF_NORTEXCH,          ILLF_NORTEXCH   },
                { NULL,         0,                      0               }
        };
        char ill_name[LIFNAMSIZ];
        int cnt;
        char *typebuf;
        char sbuf[DEFCOLS];
        int ipver = illcb->ill_ipversion;

        if (ipver != 0) {
                if ((ipver == IPV4_VERSION && ill->ill_isv6) ||
                    (ipver == IPV6_VERSION && !ill->ill_isv6)) {
                        return (WALK_NEXT);
                }
        }
        if (mdb_vread(&phyi, sizeof (phyint_t),
            (uintptr_t)ill->ill_phyint) == -1) {
```

```
2597                 mdb_warn("failed to read ill_phyint at %p",
2598                     (uintptr_t)ill->ill_phyint);
2599                 return (WALK_NEXT);
2600         }
2601         (void) mdb_readstr(ill_name, MIN(LIFNAMSIZ, ill->ill_name_length),
2602             (uintptr_t)ill->ill_name);

2604         switch (ill->ill_type) {
2605         case 0:
2606                 typebuf = "LOOPBACK";
2607                 break;
2608         case IFT_ETHER:
2609                 typebuf = "ETHER";
2610                 break;
2611         case IFT_OTHER:
2612                 typebuf = "OTHER";
2613                 break;
2614         default:
2615                 typebuf = NULL;
2616                 break;
2617         }
2618         cnt = ill->ill_refcnt + ill->ill_ire_cnt + ill->ill_nce_cnt +
2619             ill->ill_ilm_cnt + ill->ill_ncec_cnt;
2620         mdb_printf("%-?p %-8s %-3s ",
2621             addr, ill_name, ill->ill_isv6 ? "v6" : "v4");
2622         if (typebuf != NULL)
2623                 mdb_printf("%-10s ", typebuf);
2624         else
2625                 mdb_printf("%-10x ", ill->ill_type);
2626         if (verbose) {
2627                 mdb_printf("%-?p %-?p %-llb\n",
2628                     ill->ill_wq, ill->ill_ipst,
2629                     ill->ill_flags | phyi.phyint_flags, v_fmasks);
2630                 mdb_printf("%-?p %4d%4s %-?p\n",
2631                     ill->ill_phyint, cnt, "", ill->ill_grp);
2632                 mdb_snprintf(sbuf, sizeof (sbuf), "%*s %3s",
2633                     sizeof (uintptr_t) * 2, "", "");
2634                 mdb_printf("%s|\n%s+--> %3d %-18s "
2635                     "references from active threads\n",
2636                     sbuf, sbuf, ill->ill_refcnt, "ill_refcnt");
2637                 mdb_printf("%*s %7d %-18s ires referencing this ill\n",
2638                     strlen(sbuf), "", ill->ill_ire_cnt, "ill_ire_cnt");
2639                 mdb_printf("%*s %7d %-18s nces referencing this ill\n",
2640                     strlen(sbuf), "", ill->ill_nce_cnt, "ill_nce_cnt");
2641                 mdb_printf("%*s %7d %-18s ncecs referencing this ill\n",
2642                     strlen(sbuf), "", ill->ill_ncec_cnt, "ill_ncec_cnt");
2643                 mdb_printf("%*s %7d %-18s ilms referencing this ill\n",
2644                     strlen(sbuf), "", ill->ill_ilm_cnt, "ill_ilm_cnt");
2645         } else {
2646                 mdb_printf("%4d %-?p %-llb\n",
2647                     cnt, ill->ill_wq,
2648                     ill->ill_flags | phyi.phyint_flags, fmasks);
2649         }
2650         return (WALK_NEXT);
2651 }

2653 static int
2654 ill(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
2655 {
2656         ill_t ill_data;
2657         ill_cbdata_t id;
2658         int ipversion = 0;
2659         const char *zone_name = NULL;
2660         const char *opt_P = NULL;
2661         uint_t verbose = FALSE;
2662         ip_stack_t *ipst = NULL;
```



```
2664         if (mdb_getopts(argc, argv,
2665             'v', MDB_OPT_SETBITS, TRUE, &verbose,
2666             's', MDB_OPT_STR, &zone_name,
2667             'P', MDB_OPT_STR, &opt_P, NULL) != argc)
2668                 return (DCMD_USAGE);

2670         /* Follow the specified zone name to find a ip_stack_t*. */
2671         if (zone_name != NULL) {
2672                 ipst = zone_to_ips(zone_name);
2673                 if (ipst == NULL)
2674                         return (DCMD_USAGE);
2675         }

2677         if (opt_P != NULL) {
2678                 if (strcmp("v4", opt_P) == 0) {
2679                         ipversion = IPV4_VERSION;
2680                 } else if (strcmp("v6", opt_P) == 0) {
2681                         ipversion = IPV6_VERSION;
2682                 } else {
2683                         mdb_warn("invalid protocol '%s'\n", opt_P);
2684                         return (DCMD_USAGE);
2685                 }
2686         }

2688         id.verbose = verbose;
2689         id.ill_addr = addr;
2690         id.ill_ipversion = ipversion;
2691         id.ill_ipst = ipst;

2693         ill_header(verbose);
2694         if (flags & DCMD_ADDRSPEC) {
2695                 if (mdb_vread(&ill_data, sizeof (ill_t), addr) == -1) {
2696                         mdb_warn("failed to read ill at %p\n", addr);
2697                         return (DCMD_ERR);
2698                 }
2699                 (void) ill_format(addr, &ill_data, &id);
2700         } else {
2701                 if (mdb_walk("ill", (mdb_walk_cb_t)ill_cb, &id) == -1) {
2702                         mdb_warn("failed to walk ills\n");
2703                         return (DCMD_ERR);
2704                 }
2705         }
2706         return (DCMD_OK);
2707 }

2709 static void
2710 ill_help(void)
2711 {
2712         mdb_printf("Prints the following fields: ill ptr, name, "
2713             "IP version, count, ill type and ill flags.\n"
2714             "The count field is a sum of individual refcnts and is expanded "
2715             "with the -v option.\n\n");
2716         mdb_printf("Options:\n");
2717         mdb_printf("\t-P v4 | v6"
2718             "\tfilter ill structures for the specified protocol\n");
2719 }

2721 static int
2722 ip_list_walk_init(mdb_walk_state_t *wsp)
2723 {
2724         const ip_list_walk_arg_t *arg = wsp->walk_arg;
2725         ip_list_walk_data_t *iw;
2726         uintptr_t addr = (uintptr_t)(wsp->walk_addr + arg->off);

2728         if (wsp->walk_addr == NULL) {
```

```
2729                 mdb_warn("only local walks supported\n");
2730                 return (WALK_ERR);
2731         }
2732         if (mdb_vread(&wsp->walk_addr, sizeof (uintptr_t),
2733             addr) == -1) {
2734                 mdb_warn("failed to read list head at %p", addr);
2735                 return (WALK_ERR);
2736         }
2737         iw = mdb_alloc(sizeof (ip_list_walk_data_t), UM_SLEEP);
2738         iw->nextoff = arg->nextp_off;
2739         wsp->walk_data = iw;

2741         return (WALK_NEXT);
2742 }

2744 static int
2745 ip_list_walk_step(mdb_walk_state_t *wsp)
2746 {
2747         ip_list_walk_data_t *iw = wsp->walk_data;
2748         uintptr_t addr = wsp->walk_addr;

2750         if (addr == NULL)
2751                 return (WALK_DONE);
2752         wsp->walk_addr = addr + iw->nextoff;
2753         if (mdb_vread(&wsp->walk_addr, sizeof (uintptr_t),
2754             wsp->walk_addr) == -1) {
2755                 mdb_warn("failed to read list node at %p", addr);
2756                 return (WALK_ERR);
2757         }
2758         return (wsp->walk_callback(addr, iw, wsp->walk_cbdata));
2759 }

2761 static void
2762 ip_list_walk_fini(mdb_walk_state_t *wsp)
2763 {
2764         mdb_free(wsp->walk_data, sizeof (ip_list_walk_data_t));
2765 }

2767 static int
2768 ipif_walk_init(mdb_walk_state_t *wsp)
2769 {
2770         if (mdb_layered_walk("ill", wsp) == -1) {
2771                 mdb_warn("can't walk 'ills'");
2772                 return (WALK_ERR);
2773         }
2774         return (WALK_NEXT);
2775 }

2777 static int
2778 ipif_walk_step(mdb_walk_state_t *wsp)
2779 {
2780         if (mdb_pwalk("ipif_list", wsp->walk_callback, wsp->walk_cbdata,
2781             wsp->walk_addr) == -1) {
2782                 mdb_warn("can't walk 'ipif_list'");
2783                 return (WALK_ERR);
2784         }

2786         return (WALK_NEXT);
2787 }

2789 /* ARGSUSED */
2790 static int
2791 ipif_cb(uintptr_t addr, const ipif_walk_data_t *iw, ipif_cbdata_t *id)
2792 {
2793         ipif_t ipif;
```



```
2795         if (mdb_vread(&ipif, sizeof (ipif_t), (uintptr_t)addr) == -1) {
2796                 mdb_warn("failed to read ipif at %p", addr);
2797                 return (WALK_NEXT);
2798         }
2799         if (mdb_vread(&id->ill, sizeof (ill_t),
2800             (uintptr_t)ipif.ipif_ill) == -1) {
2801                 mdb_warn("failed to read ill at %p", ipif.ipif_ill);
2802                 return (WALK_NEXT);
2803         }
2804         (void) ipif_format((uintptr_t)addr, &ipif, id);
2805         return (WALK_NEXT);
2806 }

2808 static void
2809 ipif_header(boolean_t verbose)
2810 {
2811         if (verbose) {
2812                 mdb_printf("%-?s %-10s %-3s %-?s %-8s %-30s\n",
2813                     "ADDR", "NAME", "CNT", "ILL", "STFLAGS", "FLAGS");
2814                 mdb_printf("%s\n%s\n",
2815                     "LCLADDR", "BROADCAST");
2816                 mdb_printf("%<u>%80s%</u>\n", "");
2817         } else {
2818                 mdb_printf("%-?s %-10s %6s %-?s %-8s %-30s\n",
2819                     "ADDR", "NAME", "CNT", "ILL", "STFLAGS", "FLAGS");
2820                 mdb_printf("%s\n%<u>%80s%</u>\n", "LCLADDR", "");
2821         }
2822 }

2824 #ifdef _BIG_ENDIAN
2825 #define ip_ntohl_32(x)  ((x) & 0xffffffff)
2826 #else
2827 #define ip_ntohl_32(x)  (((uint32_t)(x) << 24) | \
2828                          (((uint32_t)(x) << 8) & 0xff0000) | \
2829                          (((uint32_t)(x) >> 8) & 0xff00) | \
2830                          ((uint32_t)(x)  >> 24))
2831 #endif

2833 int
2834 mask_to_prefixlen(int af, const in6_addr_t *addr)
2835 {
2836         int len = 0;
2837         int i;
2838         uint_t mask = 0;

2840         if (af == AF_INET6) {
2841                 for (i = 0; i < 4; i++) {
2842                         if (addr->s6_addr32[i] == 0xffffffff) {
2843                                 len += 32;
2844                         } else {
2845                                 mask = addr->s6_addr32[i];
2846                                 break;
2847                         }
2848                 }
2849         } else {
2850                 mask = V4_PART_OF_V6((*addr));
2851         }
2852         if (mask > 0)
2853                 len += (33 - mdb_ffs(ip_ntohl_32(mask)));
2854         return (len);
2855 }

2857 static int
2858 ipif_format(uintptr_t addr, const void *ipifptr, void *ipif_cb_arg)
2859 {
2860         const ipif_t *ipif = ipifptr;
```

```
2861        ipif_cbdata_t *ipifcb = ipif_cb_arg;
2862        boolean_t verbose = ipifcb->verbose;
2863        char ill_name[LIFNAMSIZ];
2864        char buf[LIFNAMSIZ];
2865        int cnt;
2866        static const mdb_bitmask_t sfmasks[] = {
2867                { "CO",         IPIF_CONDEMNED,         IPIF_CONDEMNED},
2868                { "CH",         IPIF_CHANGING,          IPIF_CHANGING},
2869                { "SL",         IPIF_SET_LINKLOCAL,     IPIF_SET_LINKLOCAL},
2870                { NULL,         0,                      0               }
2871        };
2872        static const mdb_bitmask_t fmasks[] = {
2873                { "UP",         IPIF_UP,                IPIF_UP         },
2874                { "UNN",        IPIF_UNNUMBERED,        IPIF_UNNUMBERED},
2875                { "DHCP",       IPIF_DHCPRUNNING,       IPIF_DHCPRUNNING},
2876                { "PRIV",       IPIF_PRIVATE,           IPIF_PRIVATE},
2877                { "NOXMT",      IPIF_NOXMIT,            IPIF_NOXMIT},
2878                { "NOLCL",      IPIF_NOLOCAL,           IPIF_NOLOCAL},
2879                { "DEPR",       IPIF_DEPRECATED,        IPIF_DEPRECATED},
2880                { "PREF",       IPIF_PREFERRED,         IPIF_PREFERRED},
2881                { "TEMP",       IPIF_TEMPORARY,         IPIF_TEMPORARY},
2882                { "ACONF",      IPIF_ADDRCONF,          IPIF_ADDRCONF},
2883                { "ANY",        IPIF_ANYCAST,           IPIF_ANYCAST},
2884                { "NFAIL",      IPIF_NOFAILOVER,        IPIF_NOFAILOVER},
2885                { NULL,         0,                      0               }
2886        };
2887        char flagsbuf[2 * A_CNT(fmasks)];
2888        char bitfields[A_CNT(fmasks)];
2889        char sflagsbuf[A_CNT(sfmasks)];
2890        char sbuf[DEFCOLS], addrstr[INET6_ADDRSTRLEN];
2891        int ipver = ipifcb->ipif_ipversion;
2892        int af;

2894        if (ipver != 0) {
2895                if ((ipver == IPV4_VERSION && ipifcb->ill.ill_isv6) ||
2896                    (ipver == IPV6_VERSION && !ipifcb->ill.ill_isv6)) {
2897                        return (WALK_NEXT);
2898                }
2899        }
2900        if ((mdb_readstr(ill_name, MIN(LIFNAMSIZ,
2901            ipifcb->ill.ill_name_length),
2902            (uintptr_t)ipifcb->ill.ill_name)) == -1) {
2903                mdb_warn("failed to read ill_name of ill %p\n", ipifcb->ill);
2904                return (WALK_NEXT);
2905        }
2906        if (ipif->ipif_id != 0) {
2907                mdb_snprintf(buf, LIFNAMSIZ, "%s:%d",
2908                    ill_name, ipif->ipif_id);
2909        } else {
2910                mdb_snprintf(buf, LIFNAMSIZ, "%s", ill_name);
2911        }
2912        mdb_snprintf(bitfields, sizeof (bitfields), "%s",
2913            ipif->ipif_addr_ready ? ",ADR" : "",
2914            ipif->ipif_was_up ? ",WU" : "",
2915            ipif->ipif_was_dup ? ",WD" : "");
2916        mdb_snprintf(flagsbuf, sizeof (flagsbuf), "%llb%s",
2917            ipif->ipif_flags, fmasks, bitfields);
2918        mdb_snprintf(sflagsbuf, sizeof (sflagsbuf), "%b",
2919            ipif->ipif_state_flags, sfmasks);

2921        cnt = ipif->ipif_refcnt;

2923        if (ipifcb->ill.ill_isv6) {
2924                mdb_snprintf(addrstr, sizeof (addrstr), "%N",
2925                    &ipif->ipif_v6lcl_addr);
2926                af = AF_INET6;
```



```
2927        } else {
2928                mdb_snprintf(addrstr, sizeof (addrstr), "%I",
2929                    V4_PART_OF_V6((ipif->ipif_v6lcl_addr)));
2930                af = AF_INET;
2931        }

2933        if (verbose) {
2934                mdb_printf("%-?p %-10s %3d %-?p %-8s %-30s\n",
2935                    addr, buf, cnt, ipif->ipif_ill,
2936                    sflagsbuf, flagsbuf);
2937                mdb_snprintf(sbuf, sizeof (sbuf), "%*s %12s",
2938                    sizeof (uintptr_t) * 2, "", "");
2939                mdb_printf("%s |\n%s +---> %4d %-15s "
2940                    "Active consistent reader cnt\n",
2941                    sbuf, sbuf, ipif->ipif_refcnt, "ipif_refcnt");
2942                mdb_printf("%-s/%d\n",
2943                    addrstr, mask_to_prefixlen(af, &ipif->ipif_v6net_mask));
2944                if (ipifcb->ill.ill_isv6) {
2945                        mdb_printf("%-N\n", &ipif->ipif_v6brd_addr);
2946                } else {
2947                        mdb_printf("%-I\n",
2948                            V4_PART_OF_V6((ipif->ipif_v6brd_addr)));
2949                }
2950        } else {
2951                mdb_printf("%-?p %-10s %6d %-?p %-8s %-30s\n",
2952                    addr, buf, cnt, ipif->ipif_ill,
2953                    sflagsbuf, flagsbuf);
2954                mdb_printf("%-s/%d\n",
2955                    addrstr, mask_to_prefixlen(af, &ipif->ipif_v6net_mask));
2956        }

2958        return (WALK_NEXT);
2959 }

2961 static int
2962 ipif(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
2963 {
2964        ipif_t ipif;
2965        ipif_cbdata_t id;
2966        int ipversion = 0;
2967        const char *opt_P = NULL;
2968        uint_t verbose = FALSE;

2970        if (mdb_getopts(argc, argv,
2971            'v', MDB_OPT_SETBITS, TRUE, &verbose,
2972            'P', MDB_OPT_STR, &opt_P, NULL) != argc)
2973                return (DCMD_USAGE);

2975        if (opt_P != NULL) {
2976                if (strcmp("v4", opt_P) == 0) {
2977                        ipversion = IPV4_VERSION;
2978                } else if (strcmp("v6", opt_P) == 0) {
2979                        ipversion = IPV6_VERSION;
2980                } else {
2981                        mdb_warn("invalid protocol '%s'\n", opt_P);
2982                        return (DCMD_USAGE);
2983                }
2984        }

2986        id.verbose = verbose;
2987        id.ipif_ipversion = ipversion;

2989        if (flags & DCMD_ADDRSPEC) {
2990                if (mdb_vread(&ipif, sizeof (ipif_t), addr) == -1) {
2991                        mdb_warn("failed to read ipif at %p\n", addr);
2992                        return (DCMD_ERR);
```

```
2993                 }
2994                 ipif_header(verbose);
2995                 if (mdb_vread(&id.ill, sizeof (ill_t),
2996                     (uintptr_t)ipif.ipif_ill) == -1) {
2997                         mdb_warn("failed to read ill at %p", ipif.ipif_ill);
2998                         return (WALK_NEXT);
2999                 }
3000                 return (ipif_format(addr, &ipif, &id));
3001         } else {
3002                 ipif_header(verbose);
3003                 if (mdb_walk("ipif", (mdb_walk_cb_t)ipif_cb, &id) == -1) {
3004                         mdb_warn("failed to walk ipifs\n");
3005                         return (DCMD_ERR);
3006                 }
3007         }
3008         return (DCMD_OK);
3009 }

3011 static void
3012 ipif_help(void)
3013 {
3014         mdb_printf("Prints the following fields: ipif ptr, name, "
3015             "count, ill ptr, state flags and ipif flags.\n"
3016             "The count field is a sum of individual refcnts and is expanded "
3017             "with the -v option.\n"
3018             "The flags field shows the following:"
3019             "\n\tUNN -> UNNUMBERED, DHCP -> DHCPRUNNING, PRIV -> PRIVATE, "
3020             "\n\tNOXMT -> NOXMIT, NOLCL -> NOLOCAL, DEPR -> DEPRECATED, "
3021             "\n\tPREF -> PREFERRED, TEMP -> TEMPORARY, ACONF -> ADDRCONF, "
3022             "\n\tANY -> ANYCAST, NFAIL -> NOFAILOVER, "
3023             "\n\tADR -> ipif_addr_ready, MU -> ipif_multicast_up, "
3024             "\n\tWU -> ipif_was_up, WD -> ipif_was_dup, "
3025             "JA -> ipif_joined_allhosts.\n\n");
3026         mdb_printf("Options:\n");
3027         mdb_printf("\t-P v4 | v6"
3028             "\tfilter ipif structures on ills for the specified protocol\n");
3029 }

3031 static int
3032 conn_status_walk_fanout(uintptr_t addr, mdb_walk_state_t *wsp,
3033     const char *walkname)
3034 {
3035         if (mdb_pwalk(walkname, wsp->walk_callback, wsp->walk_cbdata,
3036             addr) == -1) {
3037                 mdb_warn("couldn't walk '%s' at %p", walkname, addr);
3038                 return (WALK_ERR);
3039         }
3040         return (WALK_NEXT);
3041 }

3043 static int
3044 conn_status_walk_step(mdb_walk_state_t *wsp)
3045 {
3046         uintptr_t addr = wsp->walk_addr;

3048         (void) conn_status_walk_fanout(addr, wsp, "udp_hash");
3049         (void) conn_status_walk_fanout(addr, wsp, "conn_hash");
3050         (void) conn_status_walk_fanout(addr, wsp, "bind_hash");
3051         (void) conn_status_walk_fanout(addr, wsp, "proto_hash");
3052         (void) conn_status_walk_fanout(addr, wsp, "proto_v6_hash");
3053         return (WALK_NEXT);
3054 }

3056 /* ARGSUSED */
3057 static int
3058 conn_status_cb(uintptr_t addr, const void *walk_data,
```



```
3059     void *private)
3060 {
3061         netstack_t nss;
3062         char src_addrstr[INET6_ADDRSTRLEN];
3063         char rem_addrstr[INET6_ADDRSTRLEN];
3064         const ipcl_hash_walk_data_t *iw = walk_data;
3065         conn_t *conn = iw->conn;

3067         if (mdb_vread(conn, sizeof (conn_t), addr) == -1) {
3068                 mdb_warn("failed to read conn_t at %p", addr);
3069                 return (WALK_ERR);
3070         }
3071         if (mdb_vread(&nss, sizeof (nss),
3072             (uintptr_t)conn->conn_netstack) == -1) {
3073                 mdb_warn("failed to read netstack_t %p",
3074                     conn->conn_netstack);
3075                 return (WALK_ERR);
3076         }
3077         mdb_printf("%-?p %-?p %?d %?d\n", addr, conn->conn_wq,
3078             nss.netstack_stackid, conn->conn_zoneid);

3080         if (conn->conn_family == AF_INET6) {
3081                 mdb_snprintf(src_addrstr, sizeof (rem_addrstr), "%N",
3082                     &conn->conn_laddr_v6);
3083                 mdb_snprintf(rem_addrstr, sizeof (rem_addrstr), "%N",
3084                     &conn->conn_faddr_v6);
3085         } else {
3086                 mdb_snprintf(src_addrstr, sizeof (src_addrstr), "%I",
3087                     V4_PART_OF_V6((conn->conn_laddr_v6)));
3088                 mdb_snprintf(rem_addrstr, sizeof (rem_addrstr), "%I",
3089                     V4_PART_OF_V6((conn->conn_faddr_v6)));
3090         }
3091         mdb_printf("%s:%-5d\n%s:%-5d\n",
3092             src_addrstr, conn->conn_lport, rem_addrstr, conn->conn_fport);
3093         return (WALK_NEXT);
3094 }

3096 static void
3097 conn_header(void)
3098 {
3099         mdb_printf("%-?s %-?s %?s %?s\n%s\n%s\n",
3100             "ADDR", "WQ", "STACK", "ZONE", "SRC:PORT", "DEST:PORT");
3101         mdb_printf("%<u>%80s%</u>\n", "");
3102 }

3104 /*ARGSUSED*/
3105 static int
3106 conn_status(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
3107 {
3108         conn_header();
3109         if (flags & DCMD_ADDRSPEC) {
3110                 (void) conn_status_cb(addr, NULL, NULL);
3111         } else {
3112                 if (mdb_walk("conn_status", (mdb_walk_cb_t)conn_status_cb,
3113                     NULL) == -1) {
3114                         mdb_warn("failed to walk conn_fanout");
3115                         return (DCMD_ERR);
3116                 }
3117         }
3118         return (DCMD_OK);
3119 }

3121 static void
3122 conn_status_help(void)
3123 {
3124         mdb_printf("Prints conn_t structures from the following hash tables: "
```

```
3125             "\n\tips_ipcl_udp_fanout\n\tips_ipcl_bind_fanout"
3126             "\n\tips_ipcl_conn_fanout\n\tips_ipcl_proto_fanout_v4"
3127             "\n\tips_ipcl_proto_fanout_v6\n");
3128 }

3130 static int
3131 srcid_walk_step(mdb_walk_state_t *wsp)
3132 {
3133         if (mdb_pwalk("srcid_list", wsp->walk_callback, wsp->walk_cbdata,
3134             wsp->walk_addr) == -1) {
3135                 mdb_warn("can't walk 'srcid_list'");
3136                 return (WALK_ERR);
3137         }
3138         return (WALK_NEXT);
3139 }

3141 /* ARGSUSED */
3142 static int
3143 srcid_status_cb(uintptr_t addr, const void *walk_data,
3144     void *private)
3145 {
3146         srcid_map_t smp;

3148         if (mdb_vread(&smp, sizeof (srcid_map_t), addr) == -1) {
3149                 mdb_warn("failed to read srcid_map at %p", addr);
3150                 return (WALK_ERR);
3151         }
3152         mdb_printf("%-?p %3d %4d %6d %N\n",
3153             addr, smp.sm_srcid, smp.sm_zoneid, smp.sm_refcnt,
3154             &smp.sm_addr);
3155         return (WALK_NEXT);
3156 }

3158 static void
3159 srcid_header(void)
3160 {
3161         mdb_printf("%-?s %3s %4s %6s %s\n",
3162             "ADDR", "ID", "ZONE", "REFCNT", "IPADDR");
3163         mdb_printf("%<u>%80s%</u>\n", "");
3164 }

3166 /*ARGSUSED*/
3167 static int
3168 srcid_status(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
3169 {
3170         srcid_header();
3171         if (flags & DCMD_ADDRSPEC) {
3172                 (void) srcid_status_cb(addr, NULL, NULL);
3173         } else {
3174                 if (mdb_walk("srcid", (mdb_walk_cb_t)srcid_status_cb,
3175                     NULL) == -1) {
3176                         mdb_warn("failed to walk srcid_map");
3177                         return (DCMD_ERR);
3178                 }
3179         }
3180         return (DCMD_OK);
3181 }

3183 static int
3184 ilb_stacks_walk_step(mdb_walk_state_t *wsp)
3185 {
3186         return (ns_walk_step(wsp, NS_ILB));
3187 }

3189 static int
3190 ilb_rules_walk_init(mdb_walk_state_t *wsp)
```



```
3191 {
3192         ilb_stack_t ilbs;

3194         if (wsp->walk_addr == NULL)
3195                 return (WALK_ERR);

3197         if (mdb_vread(&ilbs, sizeof (ilbs), wsp->walk_addr) == -1) {
3198                 mdb_warn("failed to read ilb_stack_t at %p", wsp->walk_addr);
3199                 return (WALK_ERR);
3200         }
3201         if ((wsp->walk_addr = (uintptr_t)ilbs.ilbs_rule_head) != NULL)
3202                 return (WALK_NEXT);
3203         else
3204                 return (WALK_DONE);
3205 }

3207 static int
3208 ilb_rules_walk_step(mdb_walk_state_t *wsp)
3209 {
3210         ilb_rule_t rule;
3211         int status;

3213         if (mdb_vread(&rule, sizeof (rule), wsp->walk_addr) == -1) {
3214                 mdb_warn("failed to read ilb_rule_t at %p", wsp->walk_addr);
3215                 return (WALK_ERR);
3216         }
3217         status = wsp->walk_callback(wsp->walk_addr, &rule, wsp->walk_cbdata);
3218         if (status != WALK_NEXT)
3219                 return (status);
3220         if ((wsp->walk_addr = (uintptr_t)rule.ir_next) == NULL)
3221                 return (WALK_DONE);
3222         else
3223                 return (WALK_NEXT);
3224 }

3226 static int
3227 ilb_servers_walk_init(mdb_walk_state_t *wsp)
3228 {
3229         ilb_rule_t rule;

3231         if (wsp->walk_addr == NULL)
3232                 return (WALK_ERR);

3234         if (mdb_vread(&rule, sizeof (rule), wsp->walk_addr) == -1) {
3235                 mdb_warn("failed to read ilb_rule_t at %p", wsp->walk_addr);
3236                 return (WALK_ERR);
3237         }
3238         if ((wsp->walk_addr = (uintptr_t)rule.ir_servers) != NULL)
3239                 return (WALK_NEXT);
3240         else
3241                 return (WALK_DONE);
3242 }

3244 static int
3245 ilb_servers_walk_step(mdb_walk_state_t *wsp)
3246 {
3247         ilb_server_t server;
3248         int status;

3250         if (mdb_vread(&server, sizeof (server), wsp->walk_addr) == -1) {
3251                 mdb_warn("failed to read ilb_server_t at %p", wsp->walk_addr);
3252                 return (WALK_ERR);
3253         }
3254         status = wsp->walk_callback(wsp->walk_addr, &server, wsp->walk_cbdata);
3255         if (status != WALK_NEXT)
3256                 return (status);
```

```
3257         if ((wsp->walk_addr = (uintptr_t)server.iser_next) == NULL)
3258                 return (WALK_DONE);
3259         else
3260                 return (WALK_NEXT);
3261 }

3263 /*
3264  * Helper structure for ilb_nat_src walker.  It stores the current index of the
3265  * nat src table.
3266  */
3267 typedef struct {
3268         ilb_stack_t ilbs;
3269         int idx;
3270 } ilb_walk_t;

3272 /* Copy from list.c */
3273 #define list_object(a, node)    ((void *)(((char *)node) - (a)->list_offset))

3275 static int
3276 ilb_nat_src_walk_init(mdb_walk_state_t *wsp)
3277 {
3278         int i;
3279         ilb_walk_t *ns_walk;
3280         ilb_nat_src_entry_t *entry = NULL;

3282         if (wsp->walk_addr == NULL)
3283                 return (WALK_ERR);

3285         ns_walk = mdb_alloc(sizeof (ilb_walk_t), UM_SLEEP);
3286         if (mdb_vread(&ns_walk->ilbs, sizeof (ns_walk->ilbs),
3287             wsp->walk_addr) == -1) {
3288                 mdb_warn("failed to read ilb_stack_t at %p", wsp->walk_addr);
3289                 mdb_free(ns_walk, sizeof (ilb_walk_t));
3290                 return (WALK_ERR);
3291         }

3293         if (ns_walk->ilbs.ilbs_nat_src == NULL) {
3294                 mdb_free(ns_walk, sizeof (ilb_walk_t));
3295                 return (WALK_DONE);
3296         }

3298         wsp->walk_data = ns_walk;
3299         for (i = 0; i < ns_walk->ilbs.ilbs_nat_src_hash_size; i++) {
3300                 list_t head;
3301                 char  *khead;

3303                 /* Read in the nsh_head in the i-th element of the array. */
3304                 khead = (char *)ns_walk->ilbs.ilbs_nat_src + i *
3305                     sizeof (ilb_nat_src_hash_t);
3306                 if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3307                         mdb_warn("failed to read ilbs_nat_src at %p\n", khead);
3308                         return (WALK_ERR);
3309                 }

3311                 /*
3312                  * Note that list_next points to a kernel address and we need
3313                  * to compare list_next with the kernel address of the list
3314                  * head.  So we need to calculate the address manually.
3315                  */
3316                 if ((char *)head.list_head.list_next != khead +
3317                     offsetof(list_t, list_head)) {
3318                         entry = list_object(&head, head.list_head.list_next);
3319                         break;
3320                 }
3321         }
```



```
3323         if (entry == NULL)
3324                 return (WALK_DONE);

3326         wsp->walk_addr = (uintptr_t)entry;
3327         ns_walk->idx = i;
3328         return (WALK_NEXT);
3329 }

3331 static int
3332 ilb_nat_src_walk_step(mdb_walk_state_t *wsp)
3333 {
3334         int status;
3335         ilb_nat_src_entry_t entry, *next_entry;
3336         ilb_walk_t *ns_walk;
3337         ilb_stack_t *ilbs;
3338         list_t head;
3339         char *khead;
3340         int i;

3342         if (mdb_vread(&entry, sizeof (ilb_nat_src_entry_t),
3343             wsp->walk_addr) == -1) {
3344                 mdb_warn("failed to read ilb_nat_src_entry_t at %p",
3345                     wsp->walk_addr);
3346                 return (WALK_ERR);
3347         }
3348         status = wsp->walk_callback(wsp->walk_addr, &entry, wsp->walk_cbdata);
3349         if (status != WALK_NEXT)
3350                 return (status);

3352         ns_walk = (ilb_walk_t *)wsp->walk_data;
3353         ilbs = &ns_walk->ilbs;
3354         i = ns_walk->idx;

3356         /* Read in the nsh_head in the i-th element of the array. */
3357         khead = (char *)ilbs->ilbs_nat_src + i * sizeof (ilb_nat_src_hash_t);
3358         if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3359                 mdb_warn("failed to read ilbs_nat_src at %p\n", khead);
3360                 return (WALK_ERR);
3361         }

3363         /*
3364          * Check if there is still entry in the current list.
3365          *
3366          * Note that list_next points to a kernel address and we need to
3367          * compare list_next with the kernel address of the list head.
3368          * So we need to calculate the address manually.
3369          */
3370         if ((char *)entry.nse_link.list_next != khead + offsetof(list_t,
3371             list_head)) {
3372                 wsp->walk_addr = (uintptr_t)list_object(&head,
3373                     entry.nse_link.list_next);
3374                 return (WALK_NEXT);
3375         }

3377         /* Start with the next bucket in the array. */
3378         next_entry = NULL;
3379         for (i++; i < ilbs->ilbs_nat_src_hash_size; i++) {
3380                 khead = (char *)ilbs->ilbs_nat_src + i *
3381                     sizeof (ilb_nat_src_hash_t);
3382                 if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3383                         mdb_warn("failed to read ilbs_nat_src at %p\n", khead);
3384                         return (WALK_ERR);
3385                 }

3387                 if ((char *)head.list_head.list_next != khead +
3388                     offsetof(list_t, list_head)) {
```

```
3389                        next_entry = list_object(&head,
3390                            head.list_head.list_next);
3391                        break;
3392                }
3393        }

3395        if (next_entry == NULL)
3396                return (WALK_DONE);

3398        wsp->walk_addr = (uintptr_t)next_entry;
3399        ns_walk->idx = i;
3400        return (WALK_NEXT);
3401 }

3403 static void
3404 ilb_common_walk_fini(mdb_walk_state_t *wsp)
3405 {
3406        ilb_walk_t *walk;

3408        walk = (ilb_walk_t *)wsp->walk_data;
3409        if (walk == NULL)
3410                return;
3411        mdb_free(walk, sizeof (ilb_walk_t *));
3412 }

3414 static int
3415 ilb_conn_walk_init(mdb_walk_state_t *wsp)
3416 {
3417        int i;
3418        ilb_walk_t *conn_walk;
3419        ilb_conn_hash_t head;

3421        if (wsp->walk_addr == NULL)
3422                return (WALK_ERR);

3424        conn_walk = mdb_alloc(sizeof (ilb_walk_t), UM_SLEEP);
3425        if (mdb_vread(&conn_walk->ilbs, sizeof (conn_walk->ilbs),
3426            wsp->walk_addr) == -1) {
3427                mdb_warn("failed to read ilb_stack_t at %p", wsp->walk_addr);
3428                mdb_free(conn_walk, sizeof (ilb_walk_t));
3429                return (WALK_ERR);
3430        }

3432        if (conn_walk->ilbs.ilbs_c2s_conn_hash == NULL) {
3433                mdb_free(conn_walk, sizeof (ilb_walk_t));
3434                return (WALK_DONE);
3435        }

3437        wsp->walk_data = conn_walk;
3438        for (i = 0; i < conn_walk->ilbs.ilbs_conn_hash_size; i++) {
3439                char *khead;

3441                /* Read in the nsh_head in the i-th element of the array. */
3442                khead = (char *)conn_walk->ilbs.ilbs_c2s_conn_hash + i *
3443                    sizeof (ilb_conn_hash_t);
3444                if (mdb_vread(&head, sizeof (ilb_conn_hash_t),
3445                    (uintptr_t)khead) == -1) {
3446                        mdb_warn("failed to read ilbs_c2s_conn_hash at %p\n",
3447                            khead);
3448                        return (WALK_ERR);
3449                }

3451                if (head.ilb_connp != NULL)
3452                        break;
3453        }
```



```
3455        if (head.ilb_connp == NULL)
3456                return (WALK_DONE);

3458        wsp->walk_addr = (uintptr_t)head.ilb_connp;
3459        conn_walk->idx = i;
3460        return (WALK_NEXT);
3461 }

3463 static int
3464 ilb_conn_walk_step(mdb_walk_state_t *wsp)
3465 {
3466        int status;
3467        ilb_conn_t conn;
3468        ilb_walk_t *conn_walk;
3469        ilb_stack_t *ilbs;
3470        ilb_conn_hash_t head;
3471        char *khead;
3472        int i;

3474        if (mdb_vread(&conn, sizeof (ilb_conn_t), wsp->walk_addr) == -1) {
3475                mdb_warn("failed to read ilb_conn_t at %p", wsp->walk_addr);
3476                return (WALK_ERR);
3477        }

3479        status = wsp->walk_callback(wsp->walk_addr, &conn, wsp->walk_cbdata);
3480        if (status != WALK_NEXT)
3481                return (status);

3483        conn_walk = (ilb_walk_t *)wsp->walk_data;
3484        ilbs = &conn_walk->ilbs;
3485        i = conn_walk->idx;

3487        /* Check if there is still entry in the current list. */
3488        if (conn.conn_c2s_next != NULL) {
3489                wsp->walk_addr = (uintptr_t)conn.conn_c2s_next;
3490                return (WALK_NEXT);
3491        }

3493        /* Start with the next bucket in the array. */
3494        for (i++; i < ilbs->ilbs_conn_hash_size; i++) {
3495                khead = (char *)ilbs->ilbs_c2s_conn_hash + i *
3496                    sizeof (ilb_conn_hash_t);
3497                if (mdb_vread(&head, sizeof (ilb_conn_hash_t),
3498                    (uintptr_t)khead) == -1) {
3499                        mdb_warn("failed to read ilbs_c2s_conn_hash at %p\n",
3500                            khead);
3501                        return (WALK_ERR);
3502                }

3504                if (head.ilb_connp != NULL)
3505                        break;
3506        }

3508        if (head.ilb_connp == NULL)
3509                return (WALK_DONE);

3511        wsp->walk_addr = (uintptr_t)head.ilb_connp;
3512        conn_walk->idx = i;
3513        return (WALK_NEXT);
3514 }

3516 static int
3517 ilb_sticky_walk_init(mdb_walk_state_t *wsp)
3518 {
3519        int i;
3520        ilb_walk_t *sticky_walk;
```

```
3521         ilb_sticky_t *st = NULL;

3523         if (wsp->walk_addr == NULL)
3524                 return (WALK_ERR);

3526         sticky_walk = mdb_alloc(sizeof (ilb_walk_t), UM_SLEEP);
3527         if (mdb_vread(&sticky_walk->ilbs, sizeof (sticky_walk->ilbs),
3528             wsp->walk_addr) == -1) {
3529                 mdb_warn("failed to read ilb_stack_t at %p", wsp->walk_addr);
3530                 mdb_free(sticky_walk, sizeof (ilb_walk_t));
3531                 return (WALK_ERR);
3532         }

3534         if (sticky_walk->ilbs.ilbs_sticky_hash == NULL) {
3535                 mdb_free(sticky_walk, sizeof (ilb_walk_t));
3536                 return (WALK_DONE);
3537         }

3539         wsp->walk_data = sticky_walk;
3540         for (i = 0; i < sticky_walk->ilbs.ilbs_sticky_hash_size; i++) {
3541                 list_t head;
3542                 char *khead;

3544                 /* Read in the nsh_head in the i-th element of the array. */
3545                 khead = (char *)sticky_walk->ilbs.ilbs_sticky_hash + i *
3546                     sizeof (ilb_sticky_hash_t);
3547                 if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3548                         mdb_warn("failed to read ilbs_sticky_hash at %p\n",
3549                             khead);
3550                         return (WALK_ERR);
3551                 }

3553                 /*
3554                  * Note that list_next points to a kernel address and we need
3555                  * to compare list_next with the kernel address of the list
3556                  * head.  So we need to calculate the address manually.
3557                  */
3558                 if ((char *)head.list_head.list_next != khead +
3559                     offsetof(list_t, list_head)) {
3560                         st = list_object(&head, head.list_head.list_next);
3561                         break;
3562                 }
3563         }

3565         if (st == NULL)
3566                 return (WALK_DONE);

3568         wsp->walk_addr = (uintptr_t)st;
3569         sticky_walk->idx = i;
3570         return (WALK_NEXT);
3571 }

3573 static int
3574 ilb_sticky_walk_step(mdb_walk_state_t *wsp)
3575 {
3576         int status;
3577         ilb_sticky_t st, *st_next;
3578         ilb_walk_t *sticky_walk;
3579         ilb_stack_t *ilbs;
3580         list_t head;
3581         char *khead;
3582         int i;

3584         if (mdb_vread(&st, sizeof (ilb_sticky_t), wsp->walk_addr) == -1) {
3585                 mdb_warn("failed to read ilb_sticky_t at %p", wsp->walk_addr);
3586                 return (WALK_ERR);
```



```
3587         }

3589         status = wsp->walk_callback(wsp->walk_addr, &st, wsp->walk_cbdata);
3590         if (status != WALK_NEXT)
3591                 return (status);

3593         sticky_walk = (ilb_walk_t *)wsp->walk_data;
3594         ilbs = &sticky_walk->ilbs;
3595         i = sticky_walk->idx;

3597         /* Read in the nsh_head in the i-th element of the array. */
3598         khead = (char *)ilbs->ilbs_sticky_hash + i * sizeof (ilb_sticky_hash_t);
3599         if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3600                 mdb_warn("failed to read ilbs_sticky_hash at %p\n", khead);
3601                 return (WALK_ERR);
3602         }

3604         /*
3605          * Check if there is still entry in the current list.
3606          *
3607          * Note that list_next points to a kernel address and we need to
3608          * compare list_next with the kernel address of the list head.
3609          * So we need to calculate the address manually.
3610          */
3611         if ((char *)st.list.list_next != khead + offsetof(list_t,
3612             list_head)) {
3613                 wsp->walk_addr = (uintptr_t)list_object(&head,
3614                     st.list.list_next);
3615                 return (WALK_NEXT);
3616         }

3618         /* Start with the next bucket in the array. */
3619         st_next = NULL;
3620         for (i++; i < ilbs->ilbs_nat_src_hash_size; i++) {
3621                 khead = (char *)ilbs->ilbs_sticky_hash + i *
3622                     sizeof (ilb_sticky_hash_t);
3623                 if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3624                         mdb_warn("failed to read ilbs_sticky_hash at %p\n",
3625                             khead);
3626                         return (WALK_ERR);
3627                 }

3629                 if ((char *)head.list_head.list_next != khead +
3630                     offsetof(list_t, list_head)) {
3631                         st_next = list_object(&head,
3632                             head.list_head.list_next);
3633                         break;
3634                 }
3635         }

3637         if (st_next == NULL)
3638                 return (WALK_DONE);

3640         wsp->walk_addr = (uintptr_t)st_next;
3641         sticky_walk->idx = i;
3642         return (WALK_NEXT);
3643 }
```

```
**********************************************************
   55005 Mon Jul  9 14:38:10 2012
new/usr/src/lib/libipadm/common/ipadm_prop.c
dccp: properties
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * This file contains routines that are used to modify/retrieve protocol or
 * interface property values. It also holds all the supported properties for
 * both IP interface and protocols in 'ipadm_prop_desc_t'. Following protocols
 * are supported: IP, IPv4, IPv6, TCP, SCTP, UDP, ICMP and DCCP.
 *
 * This file also contains walkers, which walks through the property table and
 * calls the callback function, of the form 'ipadm_prop_wfunc_t' , for every
 * property in the table.
 */

#include <unistd.h>
#include <errno.h>
#include <ctype.h>
#include <fcntl.h>
#include <strings.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/sockio.h>
#include <assert.h>
#include <libdllink.h>
#include <zone.h>
#include "libipadm_impl.h"
#include <inet/tunables.h>

#define IPADM_NONESTR           "none"
#define DEF_METRIC_VAL          0       /* default metric value */

#define A_CNT(arr)      (sizeof (arr) / sizeof (arr[0]))

static ipadm_status_t   i_ipadm_validate_if(ipadm_handle_t, const char *,
                            uint_t, uint_t);

/*
 * Callback functions to retrieve property values from the kernel. These
```



```
 * functions, when required, translate the values from the kernel to a format
 * suitable for printing. For example: boolean values will be translated
 * to on/off. They also retrieve DEFAULT, PERM and POSSIBLE values for
 * a given property.
 */
static ipadm_pd_getf_t  i_ipadm_get_prop, i_ipadm_get_ifprop_flags,
                        i_ipadm_get_mtu, i_ipadm_get_metric,
                        i_ipadm_get_usesrc, i_ipadm_get_forwarding,
                        i_ipadm_get_ecnsack, i_ipadm_get_hostmodel;

/*
 * Callback function to set property values. These functions translate the
 * values to a format suitable for kernel consumption, allocates the necessary
 * ioctl buffers and then invokes ioctl().
 */
static ipadm_pd_setf_t  i_ipadm_set_prop, i_ipadm_set_mtu,
                        i_ipadm_set_ifprop_flags,
                        i_ipadm_set_metric, i_ipadm_set_usesrc,
                        i_ipadm_set_forwarding, i_ipadm_set_eprivport,
                        i_ipadm_set_ecnsack, i_ipadm_set_hostmodel;

/* array of protocols we support */
static int protocols[] = { MOD_PROTO_IP, MOD_PROTO_RAWIP,
                           MOD_PROTO_TCP, MOD_PROTO_UDP,
                           MOD_PROTO_SCTP, MOD_PROTO_DCCP };
                           MOD_PROTO_SCTP };

/*
 * Supported IP protocol properties.
 */
static ipadm_prop_desc_t ipadm_ip_prop_table[] = {
        { "arp", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
            i_ipadm_set_ifprop_flags, i_ipadm_get_onoff,
            i_ipadm_get_ifprop_flags },

        { "forwarding", IPADMPROP_CLASS_MODIF, MOD_PROTO_IPV4, 0,
            i_ipadm_set_forwarding, i_ipadm_get_onoff,
            i_ipadm_get_forwarding },

        { "metric", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
            i_ipadm_set_metric, NULL, i_ipadm_get_metric },

        { "mtu", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
            i_ipadm_set_mtu, i_ipadm_get_mtu, i_ipadm_get_mtu },

        { "exchange_routes", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
            i_ipadm_set_ifprop_flags, i_ipadm_get_onoff,
            i_ipadm_get_ifprop_flags },

        { "usesrc", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
            i_ipadm_set_usesrc, NULL, i_ipadm_get_usesrc },

        { "ttl", IPADMPROP_CLASS_MODULE, MOD_PROTO_IPV4, 0,
            i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },

        { "forwarding", IPADMPROP_CLASS_MODIF, MOD_PROTO_IPV6, 0,
            i_ipadm_set_forwarding, i_ipadm_get_onoff,
            i_ipadm_get_forwarding },

        { "hoplimit", IPADMPROP_CLASS_MODULE, MOD_PROTO_IPV6, 0,
            i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },

        { "metric", IPADMPROP_CLASS_IF, MOD_PROTO_IPV6, 0,
            i_ipadm_set_metric, NULL, i_ipadm_get_metric },

        { "mtu", IPADMPROP_CLASS_IF, MOD_PROTO_IPV6, 0,
```

```
126             i_ipadm_set_mtu, i_ipadm_get_mtu, i_ipadm_get_mtu },

128         { "nud", IPADMPROP_CLASS_IF, MOD_PROTO_IPV6, 0,
129             i_ipadm_set_ifprop_flags, i_ipadm_get_onoff,
130             i_ipadm_get_ifprop_flags },

132         { "exchange_routes", IPADMPROP_CLASS_IF, MOD_PROTO_IPV6, 0,
133             i_ipadm_set_ifprop_flags, i_ipadm_get_onoff,
134             i_ipadm_get_ifprop_flags },

136         { "usesrc", IPADMPROP_CLASS_IF, MOD_PROTO_IPV6, 0,
137             i_ipadm_set_usesrc, NULL, i_ipadm_get_usesrc },

139         { "hostmodel", IPADMPROP_CLASS_MODULE, MOD_PROTO_IPV6, 0,
140             i_ipadm_set_hostmodel, i_ipadm_get_hostmodel,
141             i_ipadm_get_hostmodel },

143         { "hostmodel", IPADMPROP_CLASS_MODULE, MOD_PROTO_IPV4, 0,
144             i_ipadm_set_hostmodel, i_ipadm_get_hostmodel,
145             i_ipadm_get_hostmodel },

147         { NULL, 0, 0, 0, NULL, NULL, NULL }
148 };
```

*unchanged_portion_omitted_*

```
242 /* Supported DCCP protocol properties */
243 static ipadm_prop_desc_t ipadm_dccp_prop_table[] = {
244         { "extra_priv_ports", IPADMPROP_CLASS_MODULE, MOD_PROTO_DCCP,
245             IPADMPROP_MULVAL, i_ipadm_set_eprivport, i_ipadm_get_prop,
246             i_ipadm_get_prop },

248         { "largest_anon_port", IPADMPROP_CLASS_MODULE, MOD_PROTO_DCCP, 0,
249             i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },

251         { "recv_maxbuf", IPADMPROP_CLASS_MODULE, MOD_PROTO_DCCP, 0,
252             i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },

254         { "send_maxbuf", IPADMPROP_CLASS_MODULE, MOD_PROTO_DCCP, 0,
255             i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },

257         { "smallest_anon_port", IPADMPROP_CLASS_MODULE, MOD_PROTO_DCCP, 0,
258             i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },

260         { "smallest_nonpriv_port", IPADMPROP_CLASS_MODULE, MOD_PROTO_DCCP, 0,
261             i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },

263         { NULL, 0, 0, 0, NULL, NULL, NULL }
264 };

266 #endif /* ! codereview */
267 /*
268  * A dummy private property structure, used while handling private
269  * protocol properties (properties not yet supported by libipadm).
270  */
271 static ipadm_prop_desc_t        ipadm_privprop =\
272         { NULL, IPADMPROP_CLASS_MODULE, MOD_PROTO_NONE, 0,
273             i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop };

275 /*
276  * Returns the property description table, for the given protocol
277  */
278 static ipadm_prop_desc_t *
279 i_ipadm_get_propdesc_table(uint_t proto)
280 {
281         switch (proto) {
282         case MOD_PROTO_IP:
```



```
283         case MOD_PROTO_IPV4:
284         case MOD_PROTO_IPV6:
285                 return (ipadm_ip_prop_table);
286         case MOD_PROTO_RAWIP:
287                 return (ipadm_icmp_prop_table);
288         case MOD_PROTO_TCP:
289                 return (ipadm_tcp_prop_table);
290         case MOD_PROTO_UDP:
291                 return (ipadm_udp_prop_table);
292         case MOD_PROTO_SCTP:
293                 return (ipadm_sctp_prop_table);
294         case MOD_PROTO_DCCP:
295                 return (ipadm_dccp_prop_table);
296 #endif /* ! codereview */
297         }

299         return (NULL);
300 }

302 static ipadm_prop_desc_t *
303 i_ipadm_get_prop_desc(const char *pname, uint_t proto, int *errp)
304 {
305         int             err = 0;
306         boolean_t       matched_name = B_FALSE;
307         ipadm_prop_desc_t *ipdp = NULL, *ipdtbl;

309         if ((ipdtbl = i_ipadm_get_propdesc_table(proto)) == NULL) {
310                 err = EINVAL;
311                 goto ret;
312         }
313         for (ipdp = ipdtbl; ipdp->ipd_name != NULL; ipdp++) {
314                 if (strcmp(pname, ipdp->ipd_name) == 0) {
315                         matched_name = B_TRUE;
316                         if (ipdp->ipd_proto == proto)
317                                 break;
318                 }
319         }
320         if (ipdp->ipd_name == NULL) {
321                 err = ENOENT;
322                 /* if we matched name, but failed protocol check */
323                 if (matched_name)
324                         err = EPROTO;
325                 ipdp = NULL;
326         }
327 ret:
328         if (errp != NULL)
329                 *errp = err;
330         return (ipdp);
331 }

333 char *
334 ipadm_proto2str(uint_t proto)
335 {
336         switch (proto) {
337         case MOD_PROTO_IP:
338                 return ("ip");
339         case MOD_PROTO_IPV4:
340                 return ("ipv4");
341         case MOD_PROTO_IPV6:
342                 return ("ipv6");
343         case MOD_PROTO_RAWIP:
344                 return ("icmp");
345         case MOD_PROTO_TCP:
346                 return ("tcp");
347         case MOD_PROTO_UDP:
348                 return ("udp");
```

```
349         case MOD_PROTO_SCTP:
350                 return ("sctp");
351         case MOD_PROTO_DCCP:
352                 return ("dccp");
353 #endif /* ! codereview */
354         }

356         return (NULL);
357 }

359 uint_t
360 ipadm_str2proto(const char *protostr)
361 {
362         if (protostr == NULL)
363                 return (MOD_PROTO_NONE);
364         if (strcmp(protostr, "tcp") == 0)
365                 return (MOD_PROTO_TCP);
366         else if (strcmp(protostr, "udp") == 0)
367                 return (MOD_PROTO_UDP);
368         else if (strcmp(protostr, "ip") == 0)
369                 return (MOD_PROTO_IP);
370         else if (strcmp(protostr, "ipv4") == 0)
371                 return (MOD_PROTO_IPV4);
372         else if (strcmp(protostr, "ipv6") == 0)
373                 return (MOD_PROTO_IPV6);
374         else if (strcmp(protostr, "icmp") == 0)
375                 return (MOD_PROTO_RAWIP);
376         else if (strcmp(protostr, "sctp") == 0)
377                 return (MOD_PROTO_SCTP);
378         else if (strcmp(protostr, "arp") == 0)
379                 return (MOD_PROTO_IP);
380         else if (strcmp(protostr, "dccp") == 0)
381                 return (MOD_PROTO_DCCP);
382 #endif /* ! codereview */

384         return (MOD_PROTO_NONE);
385 }

387 /* ARGSUSED */
388 static ipadm_status_t
389 i_ipadm_set_mtu(ipadm_handle_t iph, const void *arg,
390     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
391 {
392         struct lifreq   lifr;
393         char            *endp;
394         uint_t          mtu;
395         int             s;
396         const char      *ifname = arg;
397         char            val[MAXPROPVALLEN];

399         /* to reset MTU first retrieve the default MTU and then set it */
400         if (flags & IPADM_OPT_DEFAULT) {
401                 ipadm_status_t  status;
402                 uint_t          size = MAXPROPVALLEN;

404                 status = i_ipadm_get_prop(iph, arg, pdp, val, &size,
405                     proto, MOD_PROP_DEFAULT);
406                 if (status != IPADM_SUCCESS)
407                         return (status);
408                 pval = val;
409         }

411         errno = 0;
412         mtu = (uint_t)strtol(pval, &endp, 10);
413         if (errno != 0 || *endp != '\0')
414                 return (IPADM_INVALID_ARG);
```



```
416         bzero(&lifr, sizeof (lifr));
417         (void) strlcpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));
418         lifr.lifr_mtu = mtu;

420         s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);
421         if (ioctl(s, SIOCSLIFMTU, (caddr_t)&lifr) < 0)
422                 return (ipadm_errno2status(errno));

424         return (IPADM_SUCCESS);
425 }

427 /* ARGSUSED */
428 static ipadm_status_t
429 i_ipadm_set_metric(ipadm_handle_t iph, const void *arg,
430     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
431 {
432         struct lifreq   lifr;
433         char            *endp;
434         int             metric;
435         const char      *ifname = arg;
436         int             s;

438         /* if we are resetting, set the value to its default value */
439         if (flags & IPADM_OPT_DEFAULT) {
440                 metric = DEF_METRIC_VAL;
441         } else {
442                 errno = 0;
443                 metric = (uint_t)strtol(pval, &endp, 10);
444                 if (errno != 0 || *endp != '\0')
445                         return (IPADM_INVALID_ARG);
446         }

448         bzero(&lifr, sizeof (lifr));
449         (void) strlcpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));
450         lifr.lifr_metric = metric;

452         s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);

454         if (ioctl(s, SIOCSLIFMETRIC, (caddr_t)&lifr) < 0)
455                 return (ipadm_errno2status(errno));

457         return (IPADM_SUCCESS);
458 }

460 /* ARGSUSED */
461 static ipadm_status_t
462 i_ipadm_set_usesrc(ipadm_handle_t iph, const void *arg,
463     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
464 {
465         struct lifreq   lifr;
466         const char      *ifname = arg;
467         int             s;
468         uint_t          ifindex = 0;

470         /* if we are resetting, set the value to its default value */
471         if (flags & IPADM_OPT_DEFAULT)
472                 pval = IPADM_NONESTR;

474         /*
475          * cannot specify logical interface name. We can also filter out other
476          * bogus interface names here itself through i_ipadm_validate_ifname().
477          */
478         if (strcmp(pval, IPADM_NONESTR) != 0 &&
479             !i_ipadm_validate_ifname(iph, pval))
480                 return (IPADM_INVALID_ARG);
```

```
482         bzero(&lifr, sizeof (lifr));
483         (void) strlcpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));

485         s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);

487         if (strcmp(pval, IPADM_NONESTR) != 0) {
488                 if ((ifindex = if_nametoindex(pval)) == 0)
489                         return (ipadm_errno2status(errno));
490                 lifr.lifr_index = ifindex;
491         } else {
492                 if (ioctl(s, SIOCGLIFUSESRC, (caddr_t)&lifr) < 0)
493                         return (ipadm_errno2status(errno));
494                 lifr.lifr_index = 0;
495         }
496         if (ioctl(s, SIOCSLIFUSESRC, (caddr_t)&lifr) < 0)
497                 return (ipadm_errno2status(errno));

499         return (IPADM_SUCCESS);
500 }

502 static struct hostmodel_strval {
503         char *esm_str;
504         ip_hostmodel_t esm_val;
505 } esm_arr[] = {
506         {"weak", IP_WEAK_ES},
507         {"src-priority", IP_SRC_PRI_ES},
508         {"strong", IP_STRONG_ES},
509         {"custom", IP_MAXVAL_ES}
510 };

512 static ip_hostmodel_t
513 i_ipadm_hostmodel_str2val(const char *pval)
514 {
515         int i;

517         for (i = 0; i < A_CNT(esm_arr); i++) {
518                 if (esm_arr[i].esm_str != NULL &&
519                     strcmp(pval, esm_arr[i].esm_str) == 0) {
520                         return (esm_arr[i].esm_val);
521                 }
522         }
523         return (IP_MAXVAL_ES);
524 }

526 static char *
527 i_ipadm_hostmodel_val2str(ip_hostmodel_t pval)
528 {
529         int i;

531         for (i = 0; i < A_CNT(esm_arr); i++) {
532                 if (esm_arr[i].esm_val == pval)
533                         return (esm_arr[i].esm_str);
534         }
535         return (NULL);
536 }

538 /* ARGSUSED */
539 static ipadm_status_t
540 i_ipadm_set_hostmodel(ipadm_handle_t iph, const void *arg,
541     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
542 {
543         ip_hostmodel_t hostmodel;
544         char val[11]; /* covers uint32_max as a string */

546         if ((flags & IPADM_OPT_DEFAULT) == 0) {
```



```
547                 hostmodel = i_ipadm_hostmodel_str2val(pval);
548                 if (hostmodel == IP_MAXVAL_ES)
549                         return (IPADM_INVALID_ARG);
550                 (void) snprintf(val, sizeof (val), "%d", hostmodel);
551                 pval = val;
552         }
553         return (i_ipadm_set_prop(iph, NULL, pdp, pval, proto, flags));
554 }

556 /* ARGSUSED */
557 static ipadm_status_t
558 i_ipadm_get_hostmodel(ipadm_handle_t iph, const void *arg,
559     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
560     uint_t valtype)
561 {
562         ip_hostmodel_t hostmodel;
563         char *cp;
564         size_t nbytes;
565         ipadm_status_t status;

567         switch (valtype) {
568         case MOD_PROP_PERM:
569                 nbytes = snprintf(buf, *bufsize, "%d", MOD_PROP_PERM_RW);
570                 break;
571         case MOD_PROP_DEFAULT:
572                 nbytes = snprintf(buf, *bufsize, "weak");
573                 break;
574         case MOD_PROP_ACTIVE:
575                 status = i_ipadm_get_prop(iph, arg, pdp, buf, bufsize, proto,
576                     valtype);
577                 if (status != IPADM_SUCCESS)
578                         return (status);
579                 bcopy(buf, &hostmodel, sizeof (hostmodel));
580                 cp = i_ipadm_hostmodel_val2str(hostmodel);
581                 nbytes = snprintf(buf, *bufsize, "%s",
582                     (cp != NULL ? cp : "?"));
583                 break;
584         case MOD_PROP_POSSIBLE:
585                 nbytes = snprintf(buf, *bufsize, "strong,src-priority,weak");
586                 break;
587         default:
588                 return (IPADM_INVALID_ARG);
589         }
590         if (nbytes >= *bufsize) {
591                 /* insufficient buffer space */
592                 *bufsize = nbytes + 1;
593                 return (IPADM_NO_BUFS);
594         }
595         return (IPADM_SUCCESS);
596 }

598 /* ARGSUSED */
599 static ipadm_status_t
600 i_ipadm_set_ifprop_flags(ipadm_handle_t iph, const void *arg,
601     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
602 {
603         ipadm_status_t  status = IPADM_SUCCESS;
604         const char      *ifname = arg;
605         uint64_t        on_flags = 0, off_flags = 0;
606         boolean_t       on = B_FALSE;
607         sa_family_t     af = (proto == MOD_PROTO_IPV6 ? AF_INET6 : AF_INET);

609         /* if we are resetting, set the value to its default value */
610         if (flags & IPADM_OPT_DEFAULT) {
611                 if (strcmp(pdp->ipd_name, "exchange_routes") == 0 ||
612                     strcmp(pdp->ipd_name, "arp") == 0 ||
```

```
613                     strcmp(pdp->ipd_name, "nud") == 0) {
614                         pval = IPADM_ONSTR;
615                 } else if (strcmp(pdp->ipd_name, "forwarding") == 0) {
616                         pval = IPADM_OFFSTR;
617                 } else {
618                         return (IPADM_PROP_UNKNOWN);
619                 }
620         }

622         if (strcmp(pval, IPADM_ONSTR) == 0)
623                 on = B_TRUE;
624         else if (strcmp(pval, IPADM_OFFSTR) == 0)
625                 on = B_FALSE;
626         else
627                 return (IPADM_INVALID_ARG);

629         if (strcmp(pdp->ipd_name, "exchange_routes") == 0) {
630                 if (on)
631                         off_flags = IFF_NORTEXCH;
632                 else
633                         on_flags = IFF_NORTEXCH;
634         } else if (strcmp(pdp->ipd_name, "arp") == 0) {
635                 if (on)
636                         off_flags = IFF_NOARP;
637                 else
638                         on_flags = IFF_NOARP;
639         } else if (strcmp(pdp->ipd_name, "nud") == 0) {
640                 if (on)
641                         off_flags = IFF_NONUD;
642                 else
643                         on_flags = IFF_NONUD;
644         } else if (strcmp(pdp->ipd_name, "forwarding") == 0) {
645                 if (on)
646                         on_flags = IFF_ROUTER;
647                 else
648                         off_flags = IFF_ROUTER;
649         }

651         if (on_flags || off_flags)  {
652                 status = i_ipadm_set_flags(iph, ifname, af, on_flags,
653                     off_flags);
654         }
655         return (status);
656 }

658 /* ARGSUSED */
659 static ipadm_status_t
660 i_ipadm_set_eprivport(ipadm_handle_t iph, const void *arg,
661     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
662 {
663         nvlist_t        *portsnvl = NULL;
664         nvpair_t        *nvp;
665         ipadm_status_t  status = IPADM_SUCCESS;
666         int             err;
667         uint_t          count = 0;

669         if (flags & IPADM_OPT_DEFAULT) {
670                 assert(pval == NULL);
671                 return (i_ipadm_set_prop(iph, arg, pdp, pval, proto, flags));
672         }

674         if ((err = ipadm_str2nvlist(pval, &portsnvl, IPADM_NORVAL)) != 0)
675                 return (ipadm_errno2status(err));

677         /* count the number of ports */
678         for (nvp = nvlist_next_nvpair(portsnvl, NULL); nvp != NULL;
```



```
679             nvp = nvlist_next_nvpair(portsnvl, nvp)) {
680                 ++count;
681         }

683         if (iph->iph_flags & IPH_INIT) {
684                 flags |= IPADM_OPT_APPEND;
685         } else if (count > 1) {
686                 /*
687                  * We allow only one port to be added, removed or
688                  * assigned at a time.
689                  *
690                  * However on reboot, while initializing protocol
691                  * properties, extra_priv_ports might have multiple
692                  * values. Only in that case we allow setting multiple
693                  * values.
694                  */
695                 nvlist_free(portsnvl);
696                 return (IPADM_INVALID_ARG);
697         }

699         for (nvp = nvlist_next_nvpair(portsnvl, NULL); nvp != NULL;
700             nvp = nvlist_next_nvpair(portsnvl, nvp)) {
701                 status = i_ipadm_set_prop(iph, arg, pdp, nvpair_name(nvp),
702                     proto, flags);
703                 if (status != IPADM_SUCCESS)
704                         break;
705         }
706         nvlist_free(portsnvl);
707         return (status);
708 }

710 /* ARGSUSED */
711 static ipadm_status_t
712 i_ipadm_set_forwarding(ipadm_handle_t iph, const void *arg,
713     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
714 {
715         const char      *ifname = arg;
716         ipadm_status_t  status;

718         /*
719          * if interface name is provided, then set forwarding using the
720          * IFF_ROUTER flag
721          */
722         if (ifname != NULL) {
723                 status = i_ipadm_set_ifprop_flags(iph, ifname, pdp, pval,
724                     proto, flags);
725         } else {
726                 char    *val = NULL;

728                 /*
729                  * if the caller is IPH_LEGACY, 'pval' already contains
730                  * numeric values.
731                  */
732                 if (!(flags & IPADM_OPT_DEFAULT) &&
733                     !(iph->iph_flags & IPH_LEGACY)) {

735                         if (strcmp(pval, IPADM_ONSTR) == 0)
736                                 val = "1";
737                         else if (strcmp(pval, IPADM_OFFSTR) == 0)
738                                 val = "0";
739                         else
740                                 return (IPADM_INVALID_ARG);
741                         pval = val;
742                 }

744                 status = i_ipadm_set_prop(iph, ifname, pdp, pval, proto, flags);
```

```
745         }
746
747         return (status);
748 }
749
750 /* ARGSUSED */
751 static ipadm_status_t
752 i_ipadm_set_ecnsack(ipadm_handle_t iph, const void *arg,
753     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
754 {
755         uint_t          i;
756         char            val[MAXPROPVALLEN];
757
758         /* if IPH_LEGACY is set, 'pval' already contains numeric values */
759         if (!(flags & IPADM_OPT_DEFAULT) && !(iph->iph_flags & IPH_LEGACY)) {
760                 for (i = 0; ecn_sack_vals[i] != NULL; i++) {
761                         if (strcmp(pval, ecn_sack_vals[i]) == 0)
762                                 break;
763                 }
764                 if (ecn_sack_vals[i] == NULL)
765                         return (IPADM_INVALID_ARG);
766                 (void) snprintf(val, MAXPROPVALLEN, "%d", i);
767                 pval = val;
768         }
769
770         return (i_ipadm_set_prop(iph, arg, pdp, pval, proto, flags));
771 }
772
773 /* ARGSUSED */
774 ipadm_status_t
775 i_ipadm_get_ecnsack(ipadm_handle_t iph, const void *arg,
776     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
777     uint_t valtype)
778 {
779         ipadm_status_t  status = IPADM_SUCCESS;
780         uint_t          i, nbytes = 0;
781
782         switch (valtype) {
783         case MOD_PROP_POSSIBLE:
784                 for (i = 0; ecn_sack_vals[i] != NULL; i++) {
785                         if (i == 0)
786                                 nbytes += snprintf(buf + nbytes,
787                                     *bufsize - nbytes, "%s", ecn_sack_vals[i]);
788                         else
789                                 nbytes += snprintf(buf + nbytes,
790                                     *bufsize - nbytes, ",%s", ecn_sack_vals[i]);
791                         if (nbytes >= *bufsize)
792                                 break;
793                 }
794                 break;
795         case MOD_PROP_PERM:
796         case MOD_PROP_DEFAULT:
797         case MOD_PROP_ACTIVE:
798                 status = i_ipadm_get_prop(iph, arg, pdp, buf, bufsize, proto,
799                     valtype);
800
801                 /*
802                  * If IPH_LEGACY is set, do not convert the value returned
803                  * from kernel,
804                  */
805                 if (iph->iph_flags & IPH_LEGACY)
806                         break;
807
808                 /*
809                  * For current and default value, convert the value returned
810                  * from kernel to more discrete representation.
```



```
811                  */
812                 if (status == IPADM_SUCCESS && (valtype == MOD_PROP_ACTIVE ||
813                     valtype == MOD_PROP_DEFAULT)) {
814                         i = atoi(buf);
815                         assert(i < 3);
816                         nbytes = snprintf(buf, *bufsize, "%s",
817                             ecn_sack_vals[i]);
818                 }
819                 break;
820         default:
821                 return (IPADM_INVALID_ARG);
822         }
823         if (nbytes >= *bufsize) {
824                 /* insufficient buffer space */
825                 *bufsize = nbytes + 1;
826                 return (IPADM_NO_BUFS);
827         }
828
829         return (status);
830 }
831
832 /* ARGSUSED */
833 static ipadm_status_t
834 i_ipadm_get_forwarding(ipadm_handle_t iph, const void *arg,
835     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
836     uint_t valtype)
837 {
838         const char      *ifname = arg;
839         ipadm_status_t  status = IPADM_SUCCESS;
840
841         /*
842          * if interface name is provided, then get forwarding status using
843          * SIOCGLIFFLAGS
844          */
845         if (ifname != NULL) {
846                 status = i_ipadm_get_ifprop_flags(iph, ifname, pdp,
847                     buf, bufsize, pdp->ipd_proto, valtype);
848         } else {
849                 status = i_ipadm_get_prop(iph, ifname, pdp, buf,
850                     bufsize, proto, valtype);
851                 /*
852                  * If IPH_LEGACY is set, do not convert the value returned
853                  * from kernel,
854                  */
855                 if (iph->iph_flags & IPH_LEGACY)
856                         goto ret;
857                 if (status == IPADM_SUCCESS && (valtype == MOD_PROP_ACTIVE ||
858                     valtype == MOD_PROP_DEFAULT)) {
859                         uint_t  val = atoi(buf);
860
861                         (void) snprintf(buf, *bufsize,
862                             (val == 1 ? IPADM_ONSTR : IPADM_OFFSTR));
863                 }
864         }
865
866 ret:
867         return (status);
868 }
869
870 /* ARGSUSED */
871 static ipadm_status_t
872 i_ipadm_get_mtu(ipadm_handle_t iph, const void *arg,
873     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
874     uint_t valtype)
875 {
876         struct lifreq   lifr;
```

```
877         const char      *ifname = arg;
878         size_t          nbytes;
879         int             s;

881         switch (valtype) {
882         case MOD_PROP_PERM:
883                 nbytes = snprintf(buf, *bufsize, "%d", MOD_PROP_PERM_RW);
884                 break;
885         case MOD_PROP_DEFAULT:
886         case MOD_PROP_POSSIBLE:
887                 return (i_ipadm_get_prop(iph, arg, pdp, buf, bufsize,
888                     proto, valtype));
889         case MOD_PROP_ACTIVE:
890                 bzero(&lifr, sizeof (lifr));
891                 (void) strlcpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));
892                 s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);

894                 if (ioctl(s, SIOCGLIFMTU, (caddr_t)&lifr) < 0)
895                         return (ipadm_errno2status(errno));
896                 nbytes = snprintf(buf, *bufsize, "%u", lifr.lifr_mtu);
897                 break;
898         default:
899                 return (IPADM_INVALID_ARG);
900         }
901         if (nbytes >= *bufsize) {
902                 /* insufficient buffer space */
903                 *bufsize = nbytes + 1;
904                 return (IPADM_NO_BUFS);
905         }
906         return (IPADM_SUCCESS);
907 }

909 /* ARGSUSED */
910 static ipadm_status_t
911 i_ipadm_get_metric(ipadm_handle_t iph, const void *arg,
912     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
913     uint_t valtype)
914 {
915         struct lifreq   lifr;
916         const char      *ifname = arg;
917         size_t          nbytes;
918         int             s, val;

920         switch (valtype) {
921         case MOD_PROP_PERM:
922                 val = MOD_PROP_PERM_RW;
923                 break;
924         case MOD_PROP_DEFAULT:
925                 val = DEF_METRIC_VAL;
926                 break;
927         case MOD_PROP_ACTIVE:
928                 bzero(&lifr, sizeof (lifr));
929                 (void) strlcpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));

931                 s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);
932                 if (ioctl(s, SIOCGLIFMETRIC, (caddr_t)&lifr) < 0)
933                         return (ipadm_errno2status(errno));
934                 val = lifr.lifr_metric;
935                 break;
936         default:
937                 return (IPADM_INVALID_ARG);
938         }
939         nbytes = snprintf(buf, *bufsize, "%d", val);
940         if (nbytes >= *bufsize) {
941                 /* insufficient buffer space */
942                 *bufsize = nbytes + 1;
```



```
943                 return (IPADM_NO_BUFS);
944         }

946         return (IPADM_SUCCESS);
947 }

949 /* ARGSUSED */
950 static ipadm_status_t
951 i_ipadm_get_usesrc(ipadm_handle_t iph, const void *arg,
952     ipadm_prop_desc_t *ipd, char *buf, uint_t *bufsize, uint_t proto,
953     uint_t valtype)
954 {
955         struct lifreq   lifr;
956         const char      *ifname = arg;
957         int             s;
958         char            if_name[IF_NAMESIZE];
959         size_t          nbytes;

961         switch (valtype) {
962         case MOD_PROP_PERM:
963                 nbytes = snprintf(buf, *bufsize, "%d", MOD_PROP_PERM_RW);
964                 break;
965         case MOD_PROP_DEFAULT:
966                 nbytes = snprintf(buf, *bufsize, "%s", IPADM_NONESTR);
967                 break;
968         case MOD_PROP_ACTIVE:
969                 bzero(&lifr, sizeof (lifr));
970                 (void) strlcpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));

972                 s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);
973                 if (ioctl(s, SIOCGLIFUSESRC, (caddr_t)&lifr) < 0)
974                         return (ipadm_errno2status(errno));
975                 if (lifr.lifr_index == 0) {
976                         /* no src address was set, so print 'none' */
977                         (void) strlcpy(if_name, IPADM_NONESTR,
978                             sizeof (if_name));
979                 } else if (if_indextoname(lifr.lifr_index, if_name) == NULL) {
980                         return (ipadm_errno2status(errno));
981                 }
982                 nbytes = snprintf(buf, *bufsize, "%s", if_name);
983                 break;
984         default:
985                 return (IPADM_INVALID_ARG);
986         }
987         if (nbytes >= *bufsize) {
988                 /* insufficient buffer space */
989                 *bufsize = nbytes + 1;
990                 return (IPADM_NO_BUFS);
991         }
992         return (IPADM_SUCCESS);
993 }

995 /* ARGSUSED */
996 static ipadm_status_t
997 i_ipadm_get_ifprop_flags(ipadm_handle_t iph, const void *arg,
998     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
999     uint_t valtype)
1000 {
1001         uint64_t        intf_flags;
1002         char            *val;
1003         size_t          nbytes;
1004         const char      *ifname = arg;
1005         sa_family_t     af;
1006         ipadm_status_t  status = IPADM_SUCCESS;

1008         switch (valtype) {
```

```
1009         case MOD_PROP_PERM:
1010                 nbytes = snprintf(buf, *bufsize, "%d", MOD_PROP_PERM_RW);
1011                 break;
1012         case MOD_PROP_DEFAULT:
1013                 if (strcmp(pdp->ipd_name, "exchange_routes") == 0 ||
1014                     strcmp(pdp->ipd_name, "arp") == 0 ||
1015                     strcmp(pdp->ipd_name, "nud") == 0) {
1016                         val = IPADM_ONSTR;
1017                 } else if (strcmp(pdp->ipd_name, "forwarding") == 0) {
1018                         val = IPADM_OFFSTR;
1019                 } else {
1020                         return (IPADM_PROP_UNKNOWN);
1021                 }
1022                 nbytes = snprintf(buf, *bufsize, "%s", val);
1023                 break;
1024         case MOD_PROP_ACTIVE:
1025                 af = (proto == MOD_PROTO_IPV6 ? AF_INET6 : AF_INET);
1026                 status = i_ipadm_get_flags(iph, ifname, af, &intf_flags);
1027                 if (status != IPADM_SUCCESS)
1028                         return (status);

1030                 val = IPADM_OFFSTR;
1031                 if (strcmp(pdp->ipd_name, "exchange_routes") == 0) {
1032                         if (!(intf_flags & IFF_NORTEXCH))
1033                                 val = IPADM_ONSTR;
1034                 } else if (strcmp(pdp->ipd_name, "forwarding") == 0) {
1035                         if (intf_flags & IFF_ROUTER)
1036                                 val = IPADM_ONSTR;
1037                 } else if (strcmp(pdp->ipd_name, "arp") == 0) {
1038                         if (!(intf_flags & IFF_NOARP))
1039                                 val = IPADM_ONSTR;
1040                 } else if (strcmp(pdp->ipd_name, "nud") == 0) {
1041                         if (!(intf_flags & IFF_NONUD))
1042                                 val = IPADM_ONSTR;
1043                 }
1044                 nbytes = snprintf(buf, *bufsize, "%s", val);
1045                 break;
1046         default:
1047                 return (IPADM_INVALID_ARG);
1048         }
1049         if (nbytes >= *bufsize) {
1050                 /* insufficient buffer space */
1051                 *bufsize = nbytes + 1;
1052                 status = IPADM_NO_BUFS;
1053         }

1055         return (status);
1056 }

1058 static void
1059 i_ipadm_perm2str(char *buf, uint_t *bufsize)
1060 {
1061         uint_t perm = atoi(buf);

1063         (void) snprintf(buf, *bufsize, "%c%c",
1064             ((perm & MOD_PROP_PERM_READ) != 0) ? 'r' : '-',
1065             ((perm & MOD_PROP_PERM_WRITE) != 0) ? 'w' : '-');
1066 }

1068 /* ARGSUSED */
1069 static ipadm_status_t
1070 i_ipadm_get_prop(ipadm_handle_t iph, const void *arg,
1071     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
1072     uint_t valtype)
1073 {
1074         ipadm_status_t  status = IPADM_SUCCESS;
```



```
1075         const char      *ifname = arg;
1076         mod_ioc_prop_t  *mip;
1077         char            *pname = pdp->ipd_name;
1078         uint_t          iocsize;

1080         /* allocate sufficient ioctl buffer to retrieve value */
1081         iocsize = sizeof (mod_ioc_prop_t) + *bufsize - 1;
1082         if ((mip = calloc(1, iocsize)) == NULL)
1083                 return (IPADM_NO_BUFS);

1085         mip->mpr_version = MOD_PROP_VERSION;
1086         mip->mpr_flags = valtype;
1087         mip->mpr_proto = proto;
1088         if (ifname != NULL) {
1089                 (void) strlcpy(mip->mpr_ifname, ifname,
1090                     sizeof (mip->mpr_ifname));
1091         }
1092         (void) strlcpy(mip->mpr_name, pname, sizeof (mip->mpr_name));
1093         mip->mpr_valsize = *bufsize;

1095         if (i_ipadm_strioctl(iph->iph_sock, SIOCGETPROP, (char *)mip,
1096             iocsize) < 0) {
1097                 if (errno == ENOENT)
1098                         status = IPADM_PROP_UNKNOWN;
1099                 else
1100                         status = ipadm_errno2status(errno);
1101         } else {
1102                 bcopy(mip->mpr_val, buf, *bufsize);
1103         }

1105         free(mip);
1106         return (status);
1107 }

1109 /*
1110  * Populates the ipmgmt_prop_arg_t based on the class of property.
1111  *
1112  * For private protocol properties, while persisting information in ipadm
1113  * data store, to ensure there is no collision of namespace between ipadm
1114  * private nvpair names (which also starts with '_', see ipadm_ipmgmt.h)
1115  * and private protocol property names, we will prepend IPADM_PRIV_PROP_PREFIX
1116  * to property names.
1117  */
1118 static void
1119 i_ipadm_populate_proparg(ipmgmt_prop_arg_t *pargp, ipadm_prop_desc_t *pdp,
1120     const char *pval, const void *object)
1121 {
1122         const struct ipadm_addrobj_s *ipaddr;
1123         uint_t          class = pdp->ipd_class;
1124         uint_t          proto = pdp->ipd_proto;

1126         (void) strlcpy(pargp->ia_pname, pdp->ipd_name,
1127             sizeof (pargp->ia_pname));
1128         if (pval != NULL)
1129                 (void) strlcpy(pargp->ia_pval, pval, sizeof (pargp->ia_pval));

1131         switch (class) {
1132         case IPADMPROP_CLASS_MODULE:
1133                 /* if it's a private property then add the prefix. */
1134                 if (pdp->ipd_name[0] == '_') {
1135                         (void) snprintf(pargp->ia_pname,
1136                             sizeof (pargp->ia_pname), "_%s", pdp->ipd_name);
1137                 }
1138                 (void) strlcpy(pargp->ia_module, object,
1139                     sizeof (pargp->ia_module));
1140                 break;
```

```
1141         case IPADMPROP_CLASS_MODIF:
1142                 /* check if object is protostr or an ifname */
1143                 if (ipadm_str2proto(object) != MOD_PROTO_NONE) {
1144                         (void) strlcpy(pargp->ia_module, object,
1145                             sizeof (pargp->ia_module));
1146                         break;
1147                 }
1148                 /* it's an interface property, fall through */
1149                 /* FALLTHRU */
1150         case IPADMPROP_CLASS_IF:
1151                 (void) strlcpy(pargp->ia_ifname, object,
1152                     sizeof (pargp->ia_ifname));
1153                 (void) strlcpy(pargp->ia_module, ipadm_proto2str(proto),
1154                     sizeof (pargp->ia_module));
1155                 break;
1156         case IPADMPROP_CLASS_ADDR:
1157                 ipaddr = object;
1158                 (void) strlcpy(pargp->ia_ifname, ipaddr->ipadm_ifname,
1159                     sizeof (pargp->ia_ifname));
1160                 (void) strlcpy(pargp->ia_aobjname, ipaddr->ipadm_aobjname,
1161                     sizeof (pargp->ia_aobjname));
1162                 break;
1163         }
1164 }

1166 /*
1167  * Common function to retrieve property value for a given interface 'ifname' or
1168  * for a given protocol 'proto'. The property name is in 'pname'.
1169  *
1170  * 'valtype' determines the type of value that will be retrieved.
1171  *      IPADM_OPT_ACTIVE -      current value of the property (active config)
1172  *      IPADM_OPT_PERSIST -     value of the property from persistent store
1173  *      IPADM_OPT_DEFAULT -     default hard coded value (boot-time value)
1174  *      IPADM_OPT_PERM -        read/write permissions for the value
1175  *      IPADM_OPT_POSSIBLE -    range of values
1176  */
1177 static ipadm_status_t
1178 i_ipadm_getprop_common(ipadm_handle_t iph, const char *ifname,
1179     const char *pname, char *buf, uint_t *bufsize, uint_t proto,
1180     uint_t valtype)
1181 {
1182         ipadm_status_t          status = IPADM_SUCCESS;
1183         ipadm_prop_desc_t       *pdp;
1184         char                    priv_propname[MAXPROPNAMELEN];
1185         boolean_t               is_if = (ifname != NULL);
1186         int                     err = 0;

1188         pdp = i_ipadm_get_prop_desc(pname, proto, &err);
1189         if (err == EPROTO)
1190                 return (IPADM_BAD_PROTOCOL);
1191         /* there are no private interface properties */
1192         if (is_if && err == ENOENT)
1193                 return (IPADM_PROP_UNKNOWN);

1195         if (pdp != NULL) {
1196                 /*
1197                  * check whether the property can be
1198                  * applied on an interface
1199                  */
1200                 if (is_if && !(pdp->ipd_class & IPADMPROP_CLASS_IF))
1201                         return (IPADM_INVALID_ARG);
1202                 /*
1203                  * check whether the property can be
1204                  * applied on a module
1205                  */
1206                 if (!is_if && !(pdp->ipd_class & IPADMPROP_CLASS_MODULE))
```



```
1207                         return (IPADM_INVALID_ARG);

1209         } else {
1210                 /* private protocol properties, pass it to kernel directly */
1211                 pdp = &ipadm_privprop;
1212                 (void) strlcpy(priv_propname, pname, sizeof (priv_propname));
1213                 pdp->ipd_name = priv_propname;
1214         }

1216         switch (valtype) {
1217         case IPADM_OPT_PERM:
1218                 status = pdp->ipd_get(iph, ifname, pdp, buf, bufsize, proto,
1219                     MOD_PROP_PERM);
1220                 if (status == IPADM_SUCCESS)
1221                         i_ipadm_perm2str(buf, bufsize);
1222                 break;
1223         case IPADM_OPT_ACTIVE:
1224                 status = pdp->ipd_get(iph, ifname, pdp, buf, bufsize, proto,
1225                     MOD_PROP_ACTIVE);
1226                 break;
1227         case IPADM_OPT_DEFAULT:
1228                 status = pdp->ipd_get(iph, ifname, pdp, buf, bufsize, proto,
1229                     MOD_PROP_DEFAULT);
1230                 break;
1231         case IPADM_OPT_POSSIBLE:
1232                 if (pdp->ipd_get_range != NULL) {
1233                         status = pdp->ipd_get_range(iph, ifname, pdp, buf,
1234                             bufsize, proto, MOD_PROP_POSSIBLE);
1235                         break;
1236                 }
1237                 buf[0] = '\0';
1238                 break;
1239         case IPADM_OPT_PERSIST:
1240                 /* retrieve from database */
1241                 if (is_if)
1242                         status = i_ipadm_get_persist_propval(iph, pdp, buf,
1243                             bufsize, ifname);
1244                 else
1245                         status = i_ipadm_get_persist_propval(iph, pdp, buf,
1246                             bufsize, ipadm_proto2str(proto));
1247                 break;
1248         default:
1249                 status = IPADM_INVALID_ARG;
1250                 break;
1251         }
1252         return (status);
1253 }

1255 /*
1256  * Get protocol property of the specified protocol.
1257  */
1258 ipadm_status_t
1259 ipadm_get_prop(ipadm_handle_t iph, const char *pname, char *buf,
1260     uint_t *bufsize, uint_t proto, uint_t valtype)
1261 {
1262         /*
1263          * validate the arguments of the function.
1264          */
1265         if (iph == NULL || pname == NULL || buf == NULL ||
1266             bufsize == NULL || *bufsize == 0) {
1267                 return (IPADM_INVALID_ARG);
1268         }
1269         /*
1270          * Do we support this proto, if not return error.
1271          */
1272         if (ipadm_proto2str(proto) == NULL)
```

```
1273                 return (IPADM_NOTSUP);

1275         return (i_ipadm_getprop_common(iph, NULL, pname, buf, bufsize,
1276             proto, valtype));
1277 }

1279 /*
1280  * Get interface property of the specified interface.
1281  */
1282 ipadm_status_t
1283 ipadm_get_ifprop(ipadm_handle_t iph, const char *ifname, const char *pname,
1284     char *buf, uint_t *bufsize, uint_t proto, uint_t valtype)
1285 {
1286         /* validate the arguments of the function. */
1287         if (iph == NULL || pname == NULL || buf == NULL ||
1288             bufsize == NULL || *bufsize == 0) {
1289                 return (IPADM_INVALID_ARG);
1290         }

1292         /* Do we support this proto, if not return error. */
1293         if (ipadm_proto2str(proto) == NULL)
1294                 return (IPADM_NOTSUP);

1296         /*
1297          * check if interface name is provided for interface property and
1298          * is valid.
1299          */
1300         if (!i_ipadm_validate_ifname(iph, ifname))
1301                 return (IPADM_INVALID_ARG);

1303         return (i_ipadm_getprop_common(iph, ifname, pname, buf, bufsize,
1304             proto, valtype));
1305 }

1307 /*
1308  * Allocates sufficient ioctl buffers and copies property name and the
1309  * value, among other things. If the flag IPADM_OPT_DEFAULT is set, then
1310  * 'pval' will be NULL and it instructs the kernel to reset the current
1311  * value to property's default value.
1312  */
1313 static ipadm_status_t
1314 i_ipadm_set_prop(ipadm_handle_t iph, const void *arg,
1315     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
1316 {
1317         ipadm_status_t  status = IPADM_SUCCESS;
1318         const char      *ifname = arg;
1319         mod_ioc_prop_t  *mip;
1320         char            *pname = pdp->ipd_name;
1321         uint_t          valsize, iocsize;
1322         uint_t          iocflags = 0;

1324         if (flags & IPADM_OPT_DEFAULT) {
1325                 iocflags |= MOD_PROP_DEFAULT;
1326         } else if (flags & IPADM_OPT_ACTIVE) {
1327                 iocflags |= MOD_PROP_ACTIVE;
1328                 if (flags & IPADM_OPT_APPEND)
1329                         iocflags |= MOD_PROP_APPEND;
1330                 else if (flags & IPADM_OPT_REMOVE)
1331                         iocflags |= MOD_PROP_REMOVE;
1332         }

1334         if (pval != NULL) {
1335                 valsize = strlen(pval);
1336                 iocsize = sizeof (mod_ioc_prop_t) + valsize - 1;
1337         } else {
1338                 valsize = 0;
```



```
1339                 iocsize = sizeof (mod_ioc_prop_t);
1340         }

1342         if ((mip = calloc(1, iocsize)) == NULL)
1343                 return (IPADM_NO_BUFS);

1345         mip->mpr_version = MOD_PROP_VERSION;
1346         mip->mpr_flags = iocflags;
1347         mip->mpr_proto = proto;
1348         if (ifname != NULL) {
1349                 (void) strlcpy(mip->mpr_ifname, ifname,
1350                     sizeof (mip->mpr_ifname));
1351         }

1353         (void) strlcpy(mip->mpr_name, pname, sizeof (mip->mpr_name));
1354         mip->mpr_valsize = valsize;
1355         if (pval != NULL)
1356                 bcopy(pval, mip->mpr_val, valsize);

1358         if (i_ipadm_strioctl(iph->iph_sock, SIOCSETPROP, (char *)mip,
1359             iocsize) < 0) {
1360                 if (errno == ENOENT)
1361                         status = IPADM_PROP_UNKNOWN;
1362                 else
1363                         status = ipadm_errno2status(errno);
1364         }
1365         free(mip);
1366         return (status);
1367 }

1369 /*
1370  * Common function for modifying both protocol/interface property.
1371  *
1372  * If:
1373  *   IPADM_OPT_PERSIST is set then the value is persisted.
1374  *   IPADM_OPT_DEFAULT is set then the default value for the property will
1375  *                     be applied.
1376  */
1377 static ipadm_status_t
1378 i_ipadm_setprop_common(ipadm_handle_t iph, const char *ifname,
1379     const char *pname, const char *buf, uint_t proto, uint_t pflags)
1380 {
1381         ipadm_status_t          status = IPADM_SUCCESS;
1382         boolean_t               persist = (pflags & IPADM_OPT_PERSIST);
1383         boolean_t               reset = (pflags & IPADM_OPT_DEFAULT);
1384         ipadm_prop_desc_t       *pdp;
1385         boolean_t               is_if = (ifname != NULL);
1386         char                    priv_propname[MAXPROPNAMELEN];
1387         int                     err = 0;

1389         /* Check that property value is within the allowed size */
1390         if (!reset && strnlen(buf, MAXPROPVALLEN) >= MAXPROPVALLEN)
1391                 return (IPADM_INVALID_ARG);

1393         pdp = i_ipadm_get_prop_desc(pname, proto, &err);
1394         if (err == EPROTO)
1395                 return (IPADM_BAD_PROTOCOL);
1396         /* there are no private interface properties */
1397         if (is_if && err == ENOENT)
1398                 return (IPADM_PROP_UNKNOWN);

1400         if (pdp != NULL) {
1401                 /* do some sanity checks */
1402                 if (is_if) {
1403                         if (!(pdp->ipd_class & IPADMPROP_CLASS_IF))
1404                                 return (IPADM_INVALID_ARG);
```

```
1405                 } else {
1406                         if (!(pdp->ipd_class & IPADMPROP_CLASS_MODULE))
1407                                 return (IPADM_INVALID_ARG);
1408                 }
1409                 /*
1410                  * if the property is not multi-valued and IPADM_OPT_APPEND or
1411                  * IPADM_OPT_REMOVE is specified, return IPADM_INVALID_ARG.
1412                  */
1413                 if (!(pdp->ipd_flags & IPADMPROP_MULVAL) && (pflags &
1414                     (IPADM_OPT_APPEND|IPADM_OPT_REMOVE))) {
1415                         return (IPADM_INVALID_ARG);
1416                 }
1417         } else {
1418                 /* private protocol property, pass it to kernel directly */
1419                 pdp = &ipadm_privprop;
1420                 (void) strlcpy(priv_propname, pname, sizeof (priv_propname));
1421                 pdp->ipd_name = priv_propname;
1422         }

1424         status = pdp->ipd_set(iph, ifname, pdp, buf, proto, pflags);
1425         if (status != IPADM_SUCCESS)
1426                 return (status);

1428         if (persist) {
1429                 if (is_if)
1430                         status = i_ipadm_persist_propval(iph, pdp, buf, ifname,
1431                             pflags);
1432                 else
1433                         status = i_ipadm_persist_propval(iph, pdp, buf,
1434                             ipadm_proto2str(proto), pflags);
1435         }
1436         return (status);
1437 }

1439 /*
1440  * Sets the property value of the specified interface
1441  */
1442 ipadm_status_t
1443 ipadm_set_ifprop(ipadm_handle_t iph, const char *ifname, const char *pname,
1444     const char *buf, uint_t proto, uint_t pflags)
1445 {
1446         boolean_t       reset = (pflags & IPADM_OPT_DEFAULT);
1447         ipadm_status_t  status;

1449         /* check for solaris.network.interface.config authorization */
1450         if (!ipadm_check_auth())
1451                 return (IPADM_EAUTH);
1452         /*
1453          * validate the arguments of the function.
1454          */
1455         if (iph == NULL || pname == NULL || (!reset && buf == NULL) ||
1456             pflags == 0 || pflags == IPADM_OPT_PERSIST ||
1457             (pflags & ~(IPADM_COMMON_OPT_MASK|IPADM_OPT_DEFAULT))) {
1458                 return (IPADM_INVALID_ARG);
1459         }

1461         /*
1462          * Do we support this protocol, if not return error.
1463          */
1464         if (ipadm_proto2str(proto) == NULL)
1465                 return (IPADM_NOTSUP);

1467         /*
1468          * Validate the interface and check if a persistent
1469          * operation is performed on a temporary object.
1470          */
```



```
1471         status = i_ipadm_validate_if(iph, ifname, proto, pflags);
1472         if (status != IPADM_SUCCESS)
1473                 return (status);

1475         return (i_ipadm_setprop_common(iph, ifname, pname, buf, proto,
1476             pflags));
1477 }

1479 /*
1480  * Sets the property value of the specified protocol.
1481  */
1482 ipadm_status_t
1483 ipadm_set_prop(ipadm_handle_t iph, const char *pname, const char *buf,
1484     uint_t proto, uint_t pflags)
1485 {
1486         boolean_t       reset = (pflags & IPADM_OPT_DEFAULT);

1488         /* check for solaris.network.interface.config authorization */
1489         if (!ipadm_check_auth())
1490                 return (IPADM_EAUTH);
1491         /*
1492          * validate the arguments of the function.
1493          */
1494         if (iph == NULL || pname == NULL ||(!reset && buf == NULL) ||
1495             pflags == 0 || pflags == IPADM_OPT_PERSIST ||
1496             (pflags & ~(IPADM_COMMON_OPT_MASK|IPADM_OPT_DEFAULT|
1497             IPADM_OPT_APPEND|IPADM_OPT_REMOVE))) {
1498                 return (IPADM_INVALID_ARG);
1499         }

1501         /*
1502          * Do we support this proto, if not return error.
1503          */
1504         if (ipadm_proto2str(proto) == NULL)
1505                 return (IPADM_NOTSUP);

1507         return (i_ipadm_setprop_common(iph, NULL, pname, buf, proto,
1508             pflags));
1509 }

1511 /* helper function for ipadm_walk_proptbl */
1512 static void
1513 i_ipadm_walk_proptbl(ipadm_prop_desc_t *pdtbl, uint_t proto, uint_t class,
1514     ipadm_prop_wfunc_t *func, void *arg)
1515 {
1516         ipadm_prop_desc_t       *pdp;

1518         for (pdp = pdtbl; pdp->ipd_name != NULL; pdp++) {
1519                 if (!(pdp->ipd_class & class))
1520                         continue;

1522                 if (proto != MOD_PROTO_NONE && !(pdp->ipd_proto & proto))
1523                         continue;

1525                 /*
1526                  * we found a class specific match, call the
1527                  * user callback function.
1528                  */
1529                 if (func(arg, pdp->ipd_name, pdp->ipd_proto) == B_FALSE)
1530                         break;
1531         }
1532 }

1534 /*
1535  * Walks through all the properties, for a given protocol and property class
1536  * (protocol or interface).
```

```
 *
 * Further if proto == MOD_PROTO_NONE, then it walks through all the supported
 * protocol property tables.
 */
ipadm_status_t
ipadm_walk_proptbl(uint_t proto, uint_t class, ipadm_prop_wfunc_t *func,
    void *arg)
{
        ipadm_prop_desc_t       *pdtbl;
        ipadm_status_t          status = IPADM_SUCCESS;
        int                     i;
        int                     count = A_CNT(protocols);

        if (func == NULL)
                return (IPADM_INVALID_ARG);

        switch (class) {
        case IPADMPROP_CLASS_ADDR:
                pdtbl = ipadm_addrprop_table;
                break;
        case IPADMPROP_CLASS_IF:
        case IPADMPROP_CLASS_MODULE:
                pdtbl = i_ipadm_get_propdesc_table(proto);
                if (pdtbl == NULL && proto != MOD_PROTO_NONE)
                        return (IPADM_INVALID_ARG);
                break;
        default:
                return (IPADM_INVALID_ARG);
        }

        if (pdtbl != NULL) {
                /*
                 * proto will be MOD_PROTO_NONE in the case of
                 * IPADMPROP_CLASS_ADDR.
                 */
                i_ipadm_walk_proptbl(pdtbl, proto, class, func, arg);
        } else {
                /* Walk thru all the protocol tables, we support */
                for (i = 0; i < count; i++) {
                        pdtbl = i_ipadm_get_propdesc_table(protocols[i]);
                        i_ipadm_walk_proptbl(pdtbl, protocols[i], class, func,
                            arg);
                }
        }
        return (status);
}

/*
 * Given a property name, walks through all the instances of a property name.
 * Some properties have two instances one for v4 interfaces and another for v6
 * interfaces. For example: MTU. MTU can have different values for v4 and v6.
 * Therefore there are two properties for 'MTU'.
 *
 * This function invokes 'func' for every instance of property 'pname'
 */
ipadm_status_t
ipadm_walk_prop(const char *pname, uint_t proto, uint_t class,
    ipadm_prop_wfunc_t *func, void *arg)
{
        ipadm_prop_desc_t       *pdtbl, *pdp;
        ipadm_status_t          status = IPADM_SUCCESS;
        boolean_t               matched = B_FALSE;

        if (pname == NULL || func == NULL)
                return (IPADM_INVALID_ARG);
```



```
        switch (class) {
        case IPADMPROP_CLASS_ADDR:
                pdtbl = ipadm_addrprop_table;
                break;
        case IPADMPROP_CLASS_IF:
        case IPADMPROP_CLASS_MODULE:
                pdtbl = i_ipadm_get_propdesc_table(proto);
                break;
        default:
                return (IPADM_INVALID_ARG);
        }

        if (pdtbl == NULL)
                return (IPADM_INVALID_ARG);

        for (pdp = pdtbl; pdp->ipd_name != NULL; pdp++) {
                if (strcmp(pname, pdp->ipd_name) != 0)
                        continue;
                if (!(pdp->ipd_proto & proto))
                        continue;
                matched = B_TRUE;
                /* we found a match, call the callback function */
                if (func(arg, pdp->ipd_name, pdp->ipd_proto) == B_FALSE)
                        break;
        }
        if (!matched)
                status = IPADM_PROP_UNKNOWN;
        return (status);
}

/* ARGSUSED */
ipadm_status_t
i_ipadm_get_onoff(ipadm_handle_t iph, const void *arg, ipadm_prop_desc_t *dp,
    char *buf, uint_t *bufsize, uint_t proto, uint_t valtype)
{
        (void) snprintf(buf, *bufsize, "%s,%s", IPADM_ONSTR, IPADM_OFFSTR);
        return (IPADM_SUCCESS);
}

/*
 * Makes a door call to ipmgmtd to retrieve the persisted property value
 */
ipadm_status_t
i_ipadm_get_persist_propval(ipadm_handle_t iph, ipadm_prop_desc_t *pdp,
    char *gbuf, uint_t *gbufsize, const void *object)
{
        ipmgmt_prop_arg_t       parg;
        ipmgmt_getprop_rval_t   rval, *rvalp;
        size_t                  nbytes;
        int                     err = 0;

        bzero(&parg, sizeof (parg));
        parg.ia_cmd = IPMGMT_CMD_GETPROP;
        i_ipadm_populate_proparg(&parg, pdp, NULL, object);

        rvalp = &rval;
        err = ipadm_door_call(iph, &parg, sizeof (parg), (void **)&rvalp,
            sizeof (rval), B_FALSE);
        if (err == 0) {
                /* assert that rvalp was not reallocated */
                assert(rvalp == &rval);

                /* 'ir_pval' contains the property value */
                nbytes = snprintf(gbuf, *gbufsize, "%s", rvalp->ir_pval);
                if (nbytes >= *gbufsize) {
                        /* insufficient buffer space */
```

```
1669                        *gbufsize = nbytes + 1;
1670                        err = ENOBUFS;
1671                }
1672        }
1673        return (ipadm_errno2status(err));
1674 }

1676 /*
1677  * Persists the property value for a given property in the data store
1678  */
1679 ipadm_status_t
1680 i_ipadm_persist_propval(ipadm_handle_t iph, ipadm_prop_desc_t *pdp,
1681     const char *pval, const void *object, uint_t flags)
1682 {
1683        ipmgmt_prop_arg_t       parg;
1684        int                     err = 0;

1686        bzero(&parg, sizeof (parg));
1687        i_ipadm_populate_proparg(&parg, pdp, pval, object);
1688        /*
1689         * Check if value to be persisted need to be appended or removed. This
1690         * is required for multi-valued property.
1691         */
1692        if (flags & IPADM_OPT_APPEND)
1693                parg.ia_flags |= IPMGMT_APPEND;
1694        if (flags & IPADM_OPT_REMOVE)
1695                parg.ia_flags |= IPMGMT_REMOVE;

1697        if (flags & (IPADM_OPT_DEFAULT|IPADM_OPT_REMOVE))
1698                parg.ia_cmd = IPMGMT_CMD_RESETPROP;
1699        else
1700                parg.ia_cmd = IPMGMT_CMD_SETPROP;

1702        err = ipadm_door_call(iph, &parg, sizeof (parg), NULL, 0, B_FALSE);

1704        /*
1705         * its fine if there were no entry in the DB to delete. The user
1706         * might be changing property value, which was not changed
1707         * persistently.
1708         */
1709        if (err == ENOENT)
1710                err = 0;
1711        return (ipadm_errno2status(err));
1712 }

1714 /*
1715  * This is called from ipadm_set_ifprop() to validate the set operation.
1716  * It does the following steps:
1717  * 1. Validates the interface name.
1718  * 2. Fails if it is an IPMP meta-interface or an underlying interface.
1719  * 3. In case of a persistent operation, verifies that the
1720  *      interface is persistent.
1721  */
1722 static ipadm_status_t
1723 i_ipadm_validate_if(ipadm_handle_t iph, const char *ifname,
1724     uint_t proto, uint_t flags)
1725 {
1726        sa_family_t     af, other_af;
1727        ipadm_status_t  status;
1728        boolean_t       p_exists;
1729        boolean_t       af_exists, other_af_exists, a_exists;

1731        /* Check if the interface name is valid. */
1732        if (!i_ipadm_validate_ifname(iph, ifname))
1733                return (IPADM_INVALID_ARG);
```



```
1735        af = (proto == MOD_PROTO_IPV6 ? AF_INET6 : AF_INET);
1736        /*
1737         * Setting properties on an IPMP meta-interface or underlying
1738         * interface is not supported.
1739         */
1740        if (i_ipadm_is_ipmp(iph, ifname) || i_ipadm_is_under_ipmp(iph, ifname))
1741                return (IPADM_NOTSUP);

1743        /* Check if interface exists in the persistent configuration. */
1744        status = i_ipadm_if_pexists(iph, ifname, af, &p_exists);
1745        if (status != IPADM_SUCCESS)
1746                return (status);

1748        /* Check if interface exists in the active configuration. */
1749        af_exists = ipadm_if_enabled(iph, ifname, af);
1750        other_af = (af == AF_INET ? AF_INET6 : AF_INET);
1751        other_af_exists = ipadm_if_enabled(iph, ifname, other_af);
1752        a_exists = (af_exists || other_af_exists);
1753        if (!a_exists && p_exists)
1754                return (IPADM_OP_DISABLE_OBJ);
1755        if (!af_exists)
1756                return (IPADM_ENXIO);

1758        /*
1759         * If a persistent operation is requested, check if the underlying
1760         * IP interface is persistent.
1761         */
1762        if ((flags & IPADM_OPT_PERSIST) && !p_exists)
1763                return (IPADM_TEMPORARY_OBJ);
1764        return (IPADM_SUCCESS);
1765 }

1767 /*
1768  * Private protocol properties namespace scheme:
1769  *
1770  * PSARC 2010/080 identified the private protocol property names to be the
1771  * leading protocol names. For e.g. tcp_strong_iss, ip_strict_src_multihoming,
1772  * et al,. However to be consistent with private data-link property names,
1773  * which starts with '_', private protocol property names will start with '_'.
1774  * For e.g. _strong_iss, _strict_src_multihoming, et al,.
1775  */

1777 /* maps new private protocol property name to the old private property name */
1778 typedef struct ipadm_oname2nname_map {
1779        char    *iom_oname;
1780        char    *iom_nname;
1781        uint_t  iom_proto;
1782 } ipadm_oname2nname_map_t;

1784 /*
1785  * IP is a special case. It isn't straight forward to derive the legacy name
1786  * from the new name and vice versa. No set standard was followed in naming
1787  * the properties and hence we need a table to capture the mapping.
1788  */
1789 static ipadm_oname2nname_map_t name_map[] = {
1790        { "arp_probe_delay",            "_arp_probe_delay",
1791          MOD_PROTO_IP },
1792        { "arp_fastprobe_delay",        "_arp_fastprobe_delay",
1793          MOD_PROTO_IP },
1794        { "arp_probe_interval",         "_arp_probe_interval",
1795          MOD_PROTO_IP },
1796        { "arp_fastprobe_interval",     "_arp_fastprobe_interval",
1797          MOD_PROTO_IP },
1798        { "arp_probe_count",            "_arp_probe_count",
1799          MOD_PROTO_IP },
1800        { "arp_fastprobe_count",        "_arp_fastprobe_count",
```

```
1801             MOD_PROTO_IP },
1802         { "arp_defend_interval",        "_arp_defend_interval",
1803             MOD_PROTO_IP },
1804         { "arp_defend_rate",            "_arp_defend_rate",
1805             MOD_PROTO_IP },
1806         { "arp_defend_period",          "_arp_defend_period",
1807             MOD_PROTO_IP },
1808         { "ndp_defend_interval",        "_ndp_defend_interval",
1809             MOD_PROTO_IP },
1810         { "ndp_defend_rate",            "_ndp_defend_rate",
1811             MOD_PROTO_IP },
1812         { "ndp_defend_period",          "_ndp_defend_period",
1813             MOD_PROTO_IP },
1814         { "igmp_max_version",           "_igmp_max_version",
1815             MOD_PROTO_IP },
1816         { "mld_max_version",            "_mld_max_version",
1817             MOD_PROTO_IP },
1818         { "ipsec_override_persocket_policy", "_ipsec_override_persocket_policy",
1819             MOD_PROTO_IP },
1820         { "ipsec_policy_log_interval",  "_ipsec_policy_log_interval",
1821             MOD_PROTO_IP },
1822         { "icmp_accept_clear_messages", "_icmp_accept_clear_messages",
1823             MOD_PROTO_IP },
1824         { "igmp_accept_clear_messages", "_igmp_accept_clear_messages",
1825             MOD_PROTO_IP },
1826         { "pim_accept_clear_messages",  "_pim_accept_clear_messages",
1827             MOD_PROTO_IP },
1828         { "ip_respond_to_echo_multicast", "_respond_to_echo_multicast",
1829             MOD_PROTO_IPV4 },
1830         { "ip_send_redirects",          "_send_redirects",
1831             MOD_PROTO_IPV4 },
1832         { "ip_forward_src_routed",      "_forward_src_routed",
1833             MOD_PROTO_IPV4 },
1834         { "ip_icmp_return_data_bytes",  "_icmp_return_data_bytes",
1835             MOD_PROTO_IPV4 },
1836         { "ip_ignore_redirect",         "_ignore_redirect",
1837             MOD_PROTO_IPV4 },
1838         { "ip_strict_dst_multihoming",  "_strict_dst_multihoming",
1839             MOD_PROTO_IPV4 },
1840         { "ip_reasm_timeout",           "_reasm_timeout",
1841             MOD_PROTO_IPV4 },
1842         { "ip_strict_src_multihoming",  "_strict_src_multihoming",
1843             MOD_PROTO_IPV4 },
1844         { "ipv4_dad_announce_interval", "_dad_announce_interval",
1845             MOD_PROTO_IPV4 },
1846         { "ipv4_icmp_return_pmtu",      "_icmp_return_pmtu",
1847             MOD_PROTO_IPV4 },
1848         { "ipv6_dad_announce_interval", "_dad_announce_interval",
1849             MOD_PROTO_IPV6 },
1850         { "ipv6_icmp_return_pmtu",      "_icmp_return_pmtu",
1851             MOD_PROTO_IPV6 },
1852         { NULL, NULL, MOD_PROTO_NONE }
1853 };

1855 /*
1856  * Following API returns a new property name in ‘nname’ for the given legacy
1857  * property name in ‘oname’.
1858  */
1859 int
1860 ipadm_legacy2new_propname(const char *oname, char *nname, uint_t nnamelen,
1861     uint_t *proto)
1862 {
1863         const char      *str;
1864         ipadm_oname2nname_map_t *ionmp;

1866         /* if it’s a public property, there is nothing to return */
```



```
1867         if (i_ipadm_get_prop_desc(oname, *proto, NULL) != NULL)
1868                 return (-1);

1870         /*
1871          * we didn’t find the ‘oname’ in the table, check if the property
1872          * name begins with a leading protocol.
1873          */
1874         str = oname;
1875         switch (*proto) {
1876         case MOD_PROTO_TCP:
1877                 if (strstr(oname, "tcp_") == oname)
1878                         str += strlen("tcp");
1879                 break;
1880         case MOD_PROTO_SCTP:
1881                 if (strstr(oname, "sctp_") == oname)
1882                         str += strlen("sctp");
1883                 break;
1884         case MOD_PROTO_UDP:
1885                 if (strstr(oname, "udp_") == oname)
1886                         str += strlen("udp");
1887                 break;
1888         case MOD_PROTO_RAWIP:
1889                 if (strstr(oname, "icmp_") == oname)
1890                         str += strlen("icmp");
1891                 break;
1892         case MOD_PROTO_IP:
1893         case MOD_PROTO_IPV4:
1894         case MOD_PROTO_IPV6:
1895                 if (strstr(oname, "ip6_") == oname) {
1896                         *proto = MOD_PROTO_IPV6;
1897                         str += strlen("ip6");
1898                 } else {
1899                         for (ionmp = name_map; ionmp->iom_oname != NULL;
1900                             ionmp++) {
1901                                 if (strcmp(oname, ionmp->iom_oname) == 0) {
1902                                         str = ionmp->iom_nname;
1903                                         *proto = ionmp->iom_proto;
1904                                         break;
1905                                 }
1906                         }
1907                         if (ionmp->iom_oname != NULL)
1908                                 break;

1910                         if (strstr(oname, "ip_") == oname) {
1911                                 *proto = MOD_PROTO_IP;
1912                                 str += strlen("ip");
1913                         }
1914                 }
1915                 break;
1916         default:
1917                 return (-1);
1918         }
1919         (void) snprintf(nname, nnamelen, "%s", str);
1920         return (0);
1921 }

1923 /*
1924  * Following API is required for ndd.c alone. To maintain backward
1925  * compatibility with ndd output, we need to print the legacy name
1926  * for the new name.
1927  */
1928 int
1929 ipadm_new2legacy_propname(const char *oname, char *nname,
1930     uint_t nnamelen, uint_t proto)
1931 {
1932         char    *prefix;
```

```
1933         ipadm_oname2nname_map_t *ionmp;

1935         /* if it's a public property, there is nothing to prepend */
1936         if (i_ipadm_get_prop_desc(oname, proto, NULL) != NULL)
1937                 return (-1);

1939         switch (proto) {
1940         case MOD_PROTO_TCP:
1941                 prefix = "tcp";
1942                 break;
1943         case MOD_PROTO_SCTP:
1944                 prefix = "sctp";
1945                 break;
1946         case MOD_PROTO_UDP:
1947                 prefix = "udp";
1948                 break;
1949         case MOD_PROTO_RAWIP:
1950                 prefix = "icmp";
1951                 break;
1952         case MOD_PROTO_IP:
1953         case MOD_PROTO_IPV4:
1954         case MOD_PROTO_IPV6:
1955                 /* handle special case for IP */
1956                 for (ionmp = name_map; ionmp->iom_oname != NULL; ionmp++) {
1957                         if (strcmp(oname, ionmp->iom_nname) == 0 &&
1958                             ionmp->iom_proto == proto) {
1959                                 (void) strlcpy(nname, ionmp->iom_oname,
1960                                     nnamelen);
1961                                 return (0);
1962                         }
1963                 }
1964                 if (proto == MOD_PROTO_IPV6)
1965                         prefix = "ip6";
1966                 else
1967                         prefix = "ip";
1968                 break;
1969         default:
1970                 return (-1);
1971         }
1972         (void) snprintf(nname, nnamelen, "%s%s", prefix, oname);
1973         return (0);
1974 }
```

```
**********************************************************
   88350 Mon Jul  9 14:38:10 2012
new/usr/src/pkg/manifests/system-header.mf
dccp: small build fix
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
# Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
#

set name=pkg.fmri value=pkg:/system/header@$(PKGVERS)
set name=pkg.description \
    value="SunOS C/C++ header files for general development of software"
set name=pkg.summary value="SunOS Header Files"
set name=info.classification value=org.opensolaris.category.2008:System/Core
set name=variant.arch value=$(ARCH)
dir path=usr group=sys
dir path=usr/include
$(i386_ONLY)dir path=usr/include/$(ARCH64)
$(i386_ONLY)dir path=usr/include/$(ARCH64)/sys
dir path=usr/include/arpa
dir path=usr/include/asm
dir path=usr/include/ast
dir path=usr/include/bsm
dir path=usr/include/dat
dir path=usr/include/des
dir path=usr/include/gssapi
dir path=usr/include/hal
$(i386_ONLY)dir path=usr/include/ia32
$(i386_ONLY)dir path=usr/include/ia32/sys
dir path=usr/include/inet
dir path=usr/include/inet/kssl
dir path=usr/include/ipp
dir path=usr/include/ipp/ipgpc
dir path=usr/include/iso
dir path=usr/include/kerberosv5
dir path=usr/include/libpolkit
dir path=usr/include/net
dir path=usr/include/netinet
dir path=usr/include/nfs
dir path=usr/include/protocols
dir path=usr/include/rpc
dir path=usr/include/rpcsvc
dir path=usr/include/sasl
dir path=usr/include/scsi
```



```
dir path=usr/include/scsi/plugins
dir path=usr/include/scsi/plugins/ses
dir path=usr/include/scsi/plugins/ses/framework
dir path=usr/include/scsi/plugins/ses/vendor
dir path=usr/include/scsi/plugins/smp
dir path=usr/include/scsi/plugins/smp/engine
dir path=usr/include/scsi/plugins/smp/framework
dir path=usr/include/security
dir path=usr/include/sharefs
dir path=usr/include/sys
dir path=usr/include/sys/av
dir path=usr/include/sys/contract
dir path=usr/include/sys/crypto
dir path=usr/include/sys/dktp
dir path=usr/include/sys/fc4
dir path=usr/include/sys/fm
dir path=usr/include/sys/fm/cpu
dir path=usr/include/sys/fm/fs
dir path=usr/include/sys/fm/io
$(sparc_ONLY)dir path=usr/include/sys/fpu
dir path=usr/include/sys/fs
dir path=usr/include/sys/hotplug
dir path=usr/include/sys/hotplug/pci
dir path=usr/include/sys/ib
dir path=usr/include/sys/ib/adapters
dir path=usr/include/sys/ib/adapters/hermon
dir path=usr/include/sys/ib/adapters/tavor
dir path=usr/include/sys/ib/clients
dir path=usr/include/sys/ib/clients/ibd
dir path=usr/include/sys/ib/clients/of
dir path=usr/include/sys/ib/clients/of/rdma
dir path=usr/include/sys/ib/clients/of/sol_ofs
dir path=usr/include/sys/ib/clients/of/sol_ucma
dir path=usr/include/sys/ib/clients/of/sol_umad
dir path=usr/include/sys/ib/clients/of/sol_uverbs
dir path=usr/include/sys/ib/ibnex
dir path=usr/include/sys/ib/ibtl
dir path=usr/include/sys/ib/ibtl/impl
dir path=usr/include/sys/ib/mgt
dir path=usr/include/sys/ib/mgt/ibmf
dir path=usr/include/sys/iso
dir path=usr/include/sys/lvm
dir path=usr/include/sys/pcmcia
dir path=usr/include/sys/proc
dir path=usr/include/sys/rsm
$(i386_ONLY)dir path=usr/include/sys/sata group=sys
dir path=usr/include/sys/scsi
dir path=usr/include/sys/scsi/adapters
dir path=usr/include/sys/scsi/conf
dir path=usr/include/sys/scsi/generic
dir path=usr/include/sys/scsi/impl
dir path=usr/include/sys/scsi/targets
dir path=usr/include/sys/sysevent
dir path=usr/include/sys/tsol
dir path=usr/include/tsol
dir path=usr/include/uuid
$(sparc_ONLY)dir path=usr/include/v7
$(sparc_ONLY)dir path=usr/include/v7/sys
$(sparc_ONLY)dir path=usr/include/v9
$(sparc_ONLY)dir path=usr/include/v9/sys
dir path=usr/include/vm
dir path=usr/platform group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,A70 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300/include
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010 group=sys
```

```
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010/include
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T12 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T4 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,SPARC-Enterprise group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Serverblade1 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-100 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1000 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1500 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-2500 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-15000 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-280R group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-480R group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-880 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V215 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V240 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V250 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V440 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V445 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V490 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V890 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-2 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-250 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-4 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise-10000 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIi-Netract group=sys
$(i386_ONLY)dir path=usr/platform/i86pc group=sys
$(i386_ONLY)dir path=usr/platform/i86pc/include
$(i386_ONLY)dir path=usr/platform/i86pc/include/sys
$(i386_ONLY)dir path=usr/platform/i86pc/include/vm
$(i386_ONLY)dir path=usr/platform/i86xpv group=sys
$(i386_ONLY)dir path=usr/platform/i86xpv/include
$(i386_ONLY)dir path=usr/platform/i86xpv/include/sys
$(i386_ONLY)dir path=usr/platform/i86xpv/include/vm
$(sparc_ONLY)dir path=usr/platform/sun4u group=sys
$(sparc_ONLY)dir path=usr/platform/sun4u/include
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/clients
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/misc
$(sparc_ONLY)dir path=usr/platform/sun4u/include/vm
$(sparc_ONLY)dir path=usr/platform/sun4v group=sys
$(sparc_ONLY)dir path=usr/platform/sun4v/include
$(sparc_ONLY)dir path=usr/platform/sun4v/include/sys
$(sparc_ONLY)dir path=usr/platform/sun4v/include/vm
dir path=usr/share
dir path=usr/share/man
dir path=usr/share/man/man3head
dir path=usr/share/man/man4
dir path=usr/share/man/man5
dir path=usr/share/man/man7i
dir path=usr/share/src group=sys
dir path=usr/share/src/uts
$(i386_ONLY)dir path=usr/share/src/uts/i86pc
$(i386_ONLY)dir path=usr/share/src/uts/i86xpv
$(sparc_ONLY)dir path=usr/share/src/uts/sun4u
$(sparc_ONLY)dir path=usr/share/src/uts/sun4v
dir path=usr/xpg4
dir path=usr/xpg4/include
$(i386_ONLY)file path=usr/include/$(ARCH64)/sys/kdi_regs.h
$(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privmregs.h
$(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privregs.h
file path=usr/include/aio.h
```



```
file path=usr/include/alloca.h
file path=usr/include/apptrace.h
file path=usr/include/apptrace_impl.h
file path=usr/include/ar.h
file path=usr/include/archives.h
file path=usr/include/arpa/ftp.h
file path=usr/include/arpa/inet.h
file path=usr/include/arpa/nameser.h
file path=usr/include/arpa/nameser_compat.h
file path=usr/include/arpa/telnet.h
file path=usr/include/arpa/tftp.h
$(i386_ONLY)file path=usr/include/asm/atomic.h
$(i386_ONLY)file path=usr/include/asm/bitmap.h
$(i386_ONLY)file path=usr/include/asm/byteorder.h
$(i386_ONLY)file path=usr/include/asm/clock.h
$(i386_ONLY)file path=usr/include/asm/cpu.h
$(i386_ONLY)file path=usr/include/asm/cpuvar.h
$(sparc_ONLY)file path=usr/include/asm/flush.h
$(i386_ONLY)file path=usr/include/asm/htable.h
$(i386_ONLY)file path=usr/include/asm/mmu.h
file path=usr/include/asm/sunddi.h
file path=usr/include/asm/thread.h
file path=usr/include/assert.h
file path=usr/include/ast/align.h
file path=usr/include/ast/ast.h
file path=usr/include/ast/ast_botch.h
file path=usr/include/ast/ast_ccode.h
file path=usr/include/ast/ast_common.h
file path=usr/include/ast/ast_dir.h
file path=usr/include/ast/ast_dirent.h
file path=usr/include/ast/ast_fcntl.h
file path=usr/include/ast/ast_float.h
file path=usr/include/ast/ast_fs.h
file path=usr/include/ast/ast_getopt.h
file path=usr/include/ast/ast_iconv.h
file path=usr/include/ast/ast_lib.h
file path=usr/include/ast/ast_limits.h
file path=usr/include/ast/ast_map.h
file path=usr/include/ast/ast_mmap.h
file path=usr/include/ast/ast_mode.h
file path=usr/include/ast/ast_namval.h
file path=usr/include/ast/ast_ndbm.h
file path=usr/include/ast/ast_nl_types.h
file path=usr/include/ast/ast_param.h
file path=usr/include/ast/ast_standards.h
file path=usr/include/ast/ast_std.h
file path=usr/include/ast/ast_stdio.h
file path=usr/include/ast/ast_sys.h
file path=usr/include/ast/ast_time.h
file path=usr/include/ast/ast_tty.h
file path=usr/include/ast/ast_version.h
file path=usr/include/ast/ast_vfork.h
file path=usr/include/ast/ast_wait.h
file path=usr/include/ast/ast_wchar.h
file path=usr/include/ast/ast_windows.h
file path=usr/include/ast/bytesex.h
file path=usr/include/ast/ccode.h
file path=usr/include/ast/cdt.h
file path=usr/include/ast/cmd.h
file path=usr/include/ast/cmdext.h
file path=usr/include/ast/debug.h
file path=usr/include/ast/dirent.h
file path=usr/include/ast/dlldefs.h
file path=usr/include/ast/dt.h
file path=usr/include/ast/endian.h
file path=usr/include/ast/error.h
```

```
260 file path=usr/include/ast/find.h
261 file path=usr/include/ast/fnmatch.h
262 file path=usr/include/ast/fnv.h
263 file path=usr/include/ast/fs3d.h
264 file path=usr/include/ast/fts.h
265 file path=usr/include/ast/ftw.h
266 file path=usr/include/ast/ftwalk.h
267 file path=usr/include/ast/getopt.h
268 file path=usr/include/ast/glob.h
269 file path=usr/include/ast/hash.h
270 file path=usr/include/ast/hashkey.h
271 file path=usr/include/ast/hashpart.h
272 file path=usr/include/ast/history.h
273 file path=usr/include/ast/iconv.h
274 file path=usr/include/ast/ip6.h
275 file path=usr/include/ast/lc.h
276 file path=usr/include/ast/ls.h
277 file path=usr/include/ast/magic.h
278 file path=usr/include/ast/magicid.h
279 file path=usr/include/ast/mc.h
280 file path=usr/include/ast/mime.h
281 file path=usr/include/ast/mnt.h
282 file path=usr/include/ast/modecanon.h
283 file path=usr/include/ast/modex.h
284 file path=usr/include/ast/namval.h
285 file path=usr/include/ast/nl_types.h
286 file path=usr/include/ast/nval.h
287 file path=usr/include/ast/option.h
288 file path=usr/include/ast/preroot.h
289 file path=usr/include/ast/proc.h
290 file path=usr/include/ast/prototyped.h
291 file path=usr/include/ast/re_comp.h
292 file path=usr/include/ast/recfmt.h
293 file path=usr/include/ast/regex.h
294 file path=usr/include/ast/regexp.h
295 file path=usr/include/ast/sfdisc.h
296 file path=usr/include/ast/sfio.h
297 file path=usr/include/ast/sfio_s.h
298 file path=usr/include/ast/sfio_t.h
299 file path=usr/include/ast/shcmd.h
300 file path=usr/include/ast/shell.h
301 file path=usr/include/ast/sig.h
302 file path=usr/include/ast/stack.h
303 file path=usr/include/ast/stak.h
304 file path=usr/include/ast/stdio.h
305 file path=usr/include/ast/stk.h
306 file path=usr/include/ast/sum.h
307 file path=usr/include/ast/swap.h
308 file path=usr/include/ast/tar.h
309 file path=usr/include/ast/times.h
310 file path=usr/include/ast/tm.h
311 file path=usr/include/ast/tmx.h
312 file path=usr/include/ast/tok.h
313 file path=usr/include/ast/tv.h
314 file path=usr/include/ast/usage.h
315 file path=usr/include/ast/vdb.h
316 file path=usr/include/ast/vecargs.h
317 file path=usr/include/ast/vmalloc.h
318 file path=usr/include/ast/wait.h
319 file path=usr/include/ast/wchar.h
320 file path=usr/include/ast/wordexp.h
321 file path=usr/include/atomic.h
322 file path=usr/include/attr.h
323 file path=usr/include/auth_attr.h
324 file path=usr/include/bsm/adt.h
325 file path=usr/include/bsm/adt_event.h
```



```
326 file path=usr/include/bsm/audit.h
327 file path=usr/include/bsm/audit_kernel.h
328 file path=usr/include/bsm/audit_kevents.h
329 file path=usr/include/bsm/audit_record.h
330 file path=usr/include/bsm/audit_uevents.h
331 file path=usr/include/bsm/devices.h
332 file path=usr/include/bsm/libbsm.h
333 file path=usr/include/config_admin.h
334 file path=usr/include/cpio.h
335 file path=usr/include/crypt.h
336 file path=usr/include/cryptoutil.h
337 file path=usr/include/ctype.h
338 file path=usr/include/curses.h
339 file path=usr/include/dat/dat.h
340 file path=usr/include/dat/dat_error.h
341 file path=usr/include/dat/dat_platform_specific.h
342 file path=usr/include/dat/dat_redirection.h
343 file path=usr/include/dat/dat_registry.h
344 file path=usr/include/dat/dat_vendor_specific.h
345 file path=usr/include/dat/udat.h
346 file path=usr/include/dat/udat_config.h
347 file path=usr/include/dat/udat_redirection.h
348 file path=usr/include/dat/udat_vendor_specific.h
349 file path=usr/include/deflt.h
350 file path=usr/include/des/des.h
351 file path=usr/include/des/desdata.h
352 file path=usr/include/des/softdes.h
353 file path=usr/include/device_info.h
354 file path=usr/include/devid.h
355 file path=usr/include/devmgmt.h
356 file path=usr/include/devpoll.h
357 file path=usr/include/dial.h
358 file path=usr/include/dirent.h
359 file path=usr/include/dlfcn.h
360 file path=usr/include/door.h
361 file path=usr/include/elf.h
362 file path=usr/include/err.h
363 file path=usr/include/errno.h
364 file path=usr/include/eti.h
365 file path=usr/include/euc.h
366 file path=usr/include/exacct.h
367 file path=usr/include/exacct_impl.h
368 file path=usr/include/exec_attr.h
369 file path=usr/include/execinfo.h
370 file path=usr/include/fatal.h
371 file path=usr/include/fcntl.h
372 file path=usr/include/float.h
373 file path=usr/include/fmtmsg.h
374 file path=usr/include/fnmatch.h
375 file path=usr/include/form.h
376 file path=usr/include/ftw.h
377 file path=usr/include/gelf.h
378 file path=usr/include/getopt.h
379 file path=usr/include/getwidth.h
380 file path=usr/include/glob.h
381 file path=usr/include/grp.h
382 file path=usr/include/gssapi/gssapi.h
383 file path=usr/include/gssapi/gssapi_ext.h
384 file path=usr/include/hal/libhal-storage.h
385 file path=usr/include/hal/libhal.h
386 $(i386_ONLY)file path=usr/include/ia32/sys/asm_linkage.h
387 $(i386_ONLY)file path=usr/include/ia32/sys/kdi_regs.h
388 $(i386_ONLY)file path=usr/include/ia32/sys/machtypes.h
389 $(i386_ONLY)file path=usr/include/ia32/sys/privmregs.h
390 $(i386_ONLY)file path=usr/include/ia32/sys/privregs.h
391 $(i386_ONLY)file path=usr/include/ia32/sys/psw.h
```

```
392 $(i386_ONLY)file path=usr/include/ia32/sys/pte.h
393 $(i386_ONLY)file path=usr/include/ia32/sys/reg.h
394 $(i386_ONLY)file path=usr/include/ia32/sys/stack.h
395 $(i386_ONLY)file path=usr/include/ia32/sys/trap.h
396 $(i386_ONLY)file path=usr/include/ia32/sys/traptrace.h
397 file path=usr/include/iconv.h
398 file path=usr/include/idmap.h
399 file path=usr/include/ieeefp.h
400 file path=usr/include/ifaddrs.h
401 file path=usr/include/inet/arp.h
402 file path=usr/include/inet/common.h
403 file path=usr/include/inet/ip.h
404 file path=usr/include/inet/ip6.h
405 file path=usr/include/inet/ip6_asp.h
406 file path=usr/include/inet/ip_arp.h
407 file path=usr/include/inet/ip_ftable.h
408 file path=usr/include/inet/ip_if.h
409 file path=usr/include/inet/ip_ire.h
410 file path=usr/include/inet/ip_multi.h
411 file path=usr/include/inet/ip_netinfo.h
412 file path=usr/include/inet/ip_rts.h
413 file path=usr/include/inet/ip_stack.h
414 file path=usr/include/inet/ipclassifier.h
415 file path=usr/include/inet/ipdrop.h
416 file path=usr/include/inet/ipnet.h
417 file path=usr/include/inet/ipp_common.h
418 file path=usr/include/inet/kssl/ksslapi.h
419 file path=usr/include/inet/led.h
420 file path=usr/include/inet/mi.h
421 file path=usr/include/inet/mib2.h
422 file path=usr/include/inet/nd.h
423 file path=usr/include/inet/optcom.h
424 file path=usr/include/inet/sctp_itf.h
425 file path=usr/include/inet/snmpcom.h
426 file path=usr/include/inet/tcp.h
427 file path=usr/include/inet/tcp_sack.h
428 file path=usr/include/inet/tcp_stack.h
429 file path=usr/include/inet/tcp_stats.h
430 file path=usr/include/inet/tunables.h
431 file path=usr/include/inet/wifi_ioctl.h
432 file path=usr/include/inttypes.h
433 file path=usr/include/ipmp.h
434 file path=usr/include/ipmp_admin.h
435 file path=usr/include/ipmp_mpathd.h
436 file path=usr/include/ipmp_query.h
437 file path=usr/include/ipp/ipgpc/ipgpc.h
438 file path=usr/include/ipp/ipp.h
439 file path=usr/include/ipp/ipp_config.h
440 file path=usr/include/ipp/ipp_impl.h
441 file path=usr/include/ipp/ippctl.h
442 file path=usr/include/iso/ctype_c99.h
443 file path=usr/include/iso/ctype_iso.h
444 file path=usr/include/iso/limits_iso.h
445 file path=usr/include/iso/locale_iso.h
446 file path=usr/include/iso/setjmp_iso.h
447 file path=usr/include/iso/signal_iso.h
448 file path=usr/include/iso/stdarg_c99.h
449 file path=usr/include/iso/stdarg_iso.h
450 file path=usr/include/iso/stddef_iso.h
451 file path=usr/include/iso/stdio_c99.h
452 file path=usr/include/iso/stdio_iso.h
453 file path=usr/include/iso/stdlib_c99.h
454 file path=usr/include/iso/stdlib_iso.h
455 file path=usr/include/iso/string_iso.h
456 file path=usr/include/iso/time_iso.h
457 file path=usr/include/iso/wchar_c99.h
```



```
458 file path=usr/include/iso/wchar_iso.h
459 file path=usr/include/iso/wctype_c99.h
460 file path=usr/include/iso/wctype_iso.h
461 file path=usr/include/iso646.h
462 file path=usr/include/kerberosv5/com_err.h
463 file path=usr/include/kerberosv5/krb5.h
464 file path=usr/include/kerberosv5/mit-sipb-copyright.h
465 file path=usr/include/kerberosv5/mit_copyright.h
466 file path=usr/include/klpd.h
467 file path=usr/include/kmfapi.h
468 file path=usr/include/kmftypes.h
469 file path=usr/include/kstat.h
470 file path=usr/include/kvm.h
471 file path=usr/include/langinfo.h
472 file path=usr/include/lastlog.h
473 file path=usr/include/lber.h
474 file path=usr/include/ldap.h
475 file path=usr/include/libcontract.h
476 file path=usr/include/libctf.h
477 file path=usr/include/libdevice.h
478 file path=usr/include/libdevinfo.h
479 file path=usr/include/libdladm.h
480 file path=usr/include/libdlbridge.h
481 file path=usr/include/libdlib.h
482 file path=usr/include/libdllink.h
483 file path=usr/include/libdlpi.h
484 file path=usr/include/libdlvlan.h
485 file path=usr/include/libelf.h
486 $(i386_ONLY)file path=usr/include/libfdisk.h
487 file path=usr/include/libfstyp.h
488 file path=usr/include/libfstyp_module.h
489 file path=usr/include/libgen.h
490 file path=usr/include/libgrubmgmt.h
491 file path=usr/include/libintl.h
492 file path=usr/include/libipmi.h
493 file path=usr/include/libipp.h
494 file path=usr/include/libnvpair.h
495 file path=usr/include/libnwam.h
496 file path=usr/include/libpolkit/libpolkit.h
497 file path=usr/include/librcm.h
498 file path=usr/include/libscf.h
499 file path=usr/include/libscf_priv.h
500 file path=usr/include/libshare.h
501 file path=usr/include/libsvm.h
502 file path=usr/include/libsysevent.h
503 file path=usr/include/libsysevent_impl.h
504 file path=usr/include/libtsnet.h
505 $(sparc_ONLY)file path=usr/include/libv12n.h
506 file path=usr/include/libw.h
507 file path=usr/include/libzfs.h
508 file path=usr/include/libzoneinfo.h
509 file path=usr/include/limits.h
510 file path=usr/include/linenum.h
511 file path=usr/include/link.h
512 file path=usr/include/listen.h
513 file path=usr/include/locale.h
514 file path=usr/include/macros.h
515 file path=usr/include/maillock.h
516 file path=usr/include/malloc.h
517 file path=usr/include/md4.h
518 file path=usr/include/md5.h
519 file path=usr/include/mdiox.h
520 file path=usr/include/mdmn_changelog.h
521 file path=usr/include/memory.h
522 file path=usr/include/menu.h
523 file path=usr/include/meta.h
```

```
524 file path=usr/include/meta_basic.h
525 file path=usr/include/meta_runtime.h
526 file path=usr/include/metacl.h
527 file path=usr/include/metad.h
528 file path=usr/include/metadyn.h
529 file path=usr/include/metamed.h
530 file path=usr/include/metamhd.h
531 file path=usr/include/mhdx.h
532 file path=usr/include/mon.h
533 file path=usr/include/monetary.h
534 file path=usr/include/mp.h
535 file path=usr/include/mqueue.h
536 file path=usr/include/mtmalloc.h
537 file path=usr/include/nan.h
538 file path=usr/include/ndbm.h
539 file path=usr/include/ndpd.h
540 file path=usr/include/net/af.h
541 file path=usr/include/net/bridge.h
542 file path=usr/include/net/if.h
543 file path=usr/include/net/if_arp.h
544 file path=usr/include/net/if_dl.h
545 file path=usr/include/net/if_types.h
546 file path=usr/include/net/pfkeyv2.h
547 file path=usr/include/net/pfpolicy.h
548 file path=usr/include/net/ppp-comp.h
549 file path=usr/include/net/ppp_defs.h
550 file path=usr/include/net/pppio.h
551 file path=usr/include/net/radix.h
552 file path=usr/include/net/route.h
553 file path=usr/include/net/trill.h
554 file path=usr/include/net/vjcompress.h
555 file path=usr/include/netconfig.h
556 file path=usr/include/netdb.h
557 file path=usr/include/netdir.h
558 file path=usr/include/netinet/arp.h
559 file path=usr/include/netinet/dccp.h
560 #endif /* ! codereview */
561 file path=usr/include/netinet/dhcp.h
562 file path=usr/include/netinet/dhcp6.h
563 file path=usr/include/netinet/icmp6.h
564 file path=usr/include/netinet/icmp_var.h
565 file path=usr/include/netinet/if_ether.h
566 file path=usr/include/netinet/igmp.h
567 file path=usr/include/netinet/igmp_var.h
568 file path=usr/include/netinet/in.h
569 file path=usr/include/netinet/in_pcb.h
570 file path=usr/include/netinet/in_systm.h
571 file path=usr/include/netinet/in_var.h
572 file path=usr/include/netinet/ip.h
573 file path=usr/include/netinet/ip6.h
574 file path=usr/include/netinet/ip_icmp.h
575 file path=usr/include/netinet/ip_mroute.h
576 file path=usr/include/netinet/ip_var.h
577 file path=usr/include/netinet/pim.h
578 file path=usr/include/netinet/sctp.h
579 file path=usr/include/netinet/tcp.h
580 file path=usr/include/netinet/tcp_debug.h
581 file path=usr/include/netinet/tcp_fsm.h
582 file path=usr/include/netinet/tcp_seq.h
583 file path=usr/include/netinet/tcp_timer.h
584 file path=usr/include/netinet/tcp_var.h
585 file path=usr/include/netinet/tcpip.h
586 file path=usr/include/netinet/udp.h
587 file path=usr/include/netinet/udp_var.h
588 file path=usr/include/netinet/vrrp.h
589 file path=usr/include/nfs/auth.h
```



```
590 file path=usr/include/nfs/export.h
591 file path=usr/include/nfs/lm.h
592 file path=usr/include/nfs/mapid.h
593 file path=usr/include/nfs/mount.h
594 file path=usr/include/nfs/nfs.h
595 file path=usr/include/nfs/nfs4.h
596 file path=usr/include/nfs/nfs4_attr.h
597 file path=usr/include/nfs/nfs4_clnt.h
598 file path=usr/include/nfs/nfs4_db_impl.h
599 file path=usr/include/nfs/nfs4_idmap_impl.h
600 file path=usr/include/nfs/nfs4_kprot.h
601 file path=usr/include/nfs/nfs_acl.h
602 file path=usr/include/nfs/nfs_clnt.h
603 file path=usr/include/nfs/nfs_cmd.h
604 file path=usr/include/nfs/nfs_log.h
605 file path=usr/include/nfs/nfs_sec.h
606 file path=usr/include/nfs/nfsid_map.h
607 file path=usr/include/nfs/nfssys.h
608 file path=usr/include/nfs/rnode.h
609 file path=usr/include/nfs/rnode4.h
610 file path=usr/include/nl_types.h
611 file path=usr/include/nlist.h
612 file path=usr/include/note.h
613 file path=usr/include/nss_common.h
614 file path=usr/include/nss_dbdefs.h
615 file path=usr/include/nss_netdir.h
616 file path=usr/include/nsswitch.h
617 file path=usr/include/panel.h
618 file path=usr/include/paths.h
619 file path=usr/include/pcsample.h
620 file path=usr/include/pfmt.h
621 file path=usr/include/pkgdev.h
622 file path=usr/include/pkginfo.h
623 file path=usr/include/pkglocs.h
624 file path=usr/include/pkgstrct.h
625 file path=usr/include/pkgtrans.h
626 file path=usr/include/poll.h
627 file path=usr/include/port.h
628 file path=usr/include/priv.h
629 file path=usr/include/proc_service.h
630 file path=usr/include/procfs.h
631 file path=usr/include/prof.h
632 file path=usr/include/prof_attr.h
633 file path=usr/include/project.h
634 file path=usr/include/protocols/dumprestore.h
635 file path=usr/include/protocols/routed.h
636 file path=usr/include/protocols/rwhod.h
637 file path=usr/include/protocols/timed.h
638 file path=usr/include/pthread.h
639 file path=usr/include/pw.h
640 file path=usr/include/pwd.h
641 file path=usr/include/rcm_module.h
642 file path=usr/include/rctl.h
643 file path=usr/include/re_comp.h
644 file path=usr/include/regex.h
645 file path=usr/include/regexp.h
646 file path=usr/include/regexpr.h
647 file path=usr/include/resolv.h
648 file path=usr/include/rje.h
649 file path=usr/include/rp_plugin.h
650 file path=usr/include/rpc/auth.h
651 file path=usr/include/rpc/auth_des.h
652 file path=usr/include/rpc/auth_sys.h
653 file path=usr/include/rpc/auth_unix.h
654 file path=usr/include/rpc/bootparam.h
655 file path=usr/include/rpc/clnt.h
```

```
656 file path=usr/include/rpc/clnt_soc.h
657 file path=usr/include/rpc/clnt_stat.h
658 file path=usr/include/rpc/des_crypt.h
659 $(sparc_ONLY)file path=usr/include/rpc/ib.h
660 file path=usr/include/rpc/key_prot.h
661 file path=usr/include/rpc/nettype.h
662 file path=usr/include/rpc/pmap_clnt.h
663 file path=usr/include/rpc/pmap_prot.h
664 file path=usr/include/rpc/pmap_prot.x
665 file path=usr/include/rpc/pmap_rmt.h
666 file path=usr/include/rpc/raw.h
667 file path=usr/include/rpc/rpc.h
668 file path=usr/include/rpc/rpc_com.h
669 file path=usr/include/rpc/rpc_msg.h
670 file path=usr/include/rpc/rpc_rdma.h
671 file path=usr/include/rpc/rpc_sztypes.h
672 file path=usr/include/rpc/rpcb_clnt.h
673 file path=usr/include/rpc/rpcb_prot.h
674 file path=usr/include/rpc/rpcb_prot.x
675 file path=usr/include/rpc/rpcent.h
676 file path=usr/include/rpc/rpcsec_gss.h
677 file path=usr/include/rpc/rpcsys.h
678 file path=usr/include/rpc/svc.h
679 file path=usr/include/rpc/svc_auth.h
680 file path=usr/include/rpc/svc_mt.h
681 file path=usr/include/rpc/svc_soc.h
682 file path=usr/include/rpc/types.h
683 file path=usr/include/rpc/xdr.h
684 file path=usr/include/rpcsvc/autofs_prot.h
685 file path=usr/include/rpcsvc/autofs_prot.x
686 file path=usr/include/rpcsvc/bootparam.h
687 file path=usr/include/rpcsvc/bootparam_prot.h
688 file path=usr/include/rpcsvc/bootparam_prot.x
689 file path=usr/include/rpcsvc/dbm.h
690 file path=usr/include/rpcsvc/key_prot.x
691 file path=usr/include/rpcsvc/mount.h
692 file path=usr/include/rpcsvc/mount.x
693 file path=usr/include/rpcsvc/nfs4_prot.h
694 file path=usr/include/rpcsvc/nfs4_prot.x
695 file path=usr/include/rpcsvc/nfs_acl.h
696 file path=usr/include/rpcsvc/nfs_acl.x
697 file path=usr/include/rpcsvc/nfs_prot.h
698 file path=usr/include/rpcsvc/nfs_prot.x
699 file path=usr/include/rpcsvc/nis.h
700 file path=usr/include/rpcsvc/nis.x
701 file path=usr/include/rpcsvc/nis_db.h
702 file path=usr/include/rpcsvc/nis_object.x
703 file path=usr/include/rpcsvc/nislib.h
704 file path=usr/include/rpcsvc/nlm_prot.h
705 file path=usr/include/rpcsvc/nlm_prot.x
706 file path=usr/include/rpcsvc/nsm_addr.h
707 file path=usr/include/rpcsvc/nsm_addr.x
708 file path=usr/include/rpcsvc/rex.h
709 file path=usr/include/rpcsvc/rex.x
710 file path=usr/include/rpcsvc/rpc_sztypes.h
711 file path=usr/include/rpcsvc/rpc_sztypes.x
712 file path=usr/include/rpcsvc/rquota.h
713 file path=usr/include/rpcsvc/rquota.x
714 file path=usr/include/rpcsvc/rstat.h
715 file path=usr/include/rpcsvc/rstat.x
716 file path=usr/include/rpcsvc/rusers.h
717 file path=usr/include/rpcsvc/rusers.x
718 file path=usr/include/rpcsvc/rwall.h
719 file path=usr/include/rpcsvc/rwall.x
720 file path=usr/include/rpcsvc/sm_inter.h
721 file path=usr/include/rpcsvc/sm_inter.x
```



```
722 file path=usr/include/rpcsvc/spray.h
723 file path=usr/include/rpcsvc/spray.x
724 file path=usr/include/rpcsvc/ufs_prot.h
725 file path=usr/include/rpcsvc/ufs_prot.x
726 file path=usr/include/rpcsvc/yp.x
727 file path=usr/include/rpcsvc/yp_prot.h
728 file path=usr/include/rpcsvc/ypclnt.h
729 file path=usr/include/rpcsvc/yppasswd.h
730 file path=usr/include/rpcsvc/ypupd.h
731 file path=usr/include/rsmapi.h
732 file path=usr/include/rtld_db.h
733 file path=usr/include/sac.h
734 file path=usr/include/sasl/prop.h
735 file path=usr/include/sasl/sasl.h
736 file path=usr/include/sasl/saslplug.h
737 file path=usr/include/sasl/saslutil.h
738 file path=usr/include/sched.h
739 file path=usr/include/schedctl.h
740 file path=usr/include/scsi/libscsi.h
741 file path=usr/include/scsi/libses.h
742 file path=usr/include/scsi/libses_plugin.h
743 file path=usr/include/scsi/libsmp.h
744 file path=usr/include/scsi/libsmp_plugin.h
745 file path=usr/include/scsi/plugins/ses/framework/libses.h
746 file path=usr/include/scsi/plugins/ses/framework/ses2.h
747 file path=usr/include/scsi/plugins/ses/framework/ses2_impl.h
748 file path=usr/include/scsi/plugins/ses/vendor/sun.h
749 file path=usr/include/sdp.h
750 file path=usr/include/search.h
751 file path=usr/include/secdb.h
752 file path=usr/include/security/auditd.h
753 file path=usr/include/security/cryptoki.h
754 file path=usr/include/security/pam_appl.h
755 file path=usr/include/security/pam_modules.h
756 file path=usr/include/security/pkcs11.h
757 file path=usr/include/security/pkcs11f.h
758 file path=usr/include/security/pkcs11t.h
759 file path=usr/include/semaphore.h
760 file path=usr/include/setjmp.h
761 file path=usr/include/sgtty.h
762 file path=usr/include/sha1.h
763 file path=usr/include/sha2.h
764 file path=usr/include/shadow.h
765 file path=usr/include/sharefs/share.h
766 file path=usr/include/sharefs/sharefs.h
767 file path=usr/include/sharefs/sharetab.h
768 file path=usr/include/siginfo.h
769 file path=usr/include/signal.h
770 file path=usr/include/sip.h
771 file path=usr/include/smbios.h
772 file path=usr/include/spawn.h
773 $(i386_ONLY)file path=usr/include/stack_unwind.h
774 file path=usr/include/stdarg.h
775 file path=usr/include/stdbool.h
776 file path=usr/include/stddef.h
777 file path=usr/include/stdint.h
778 file path=usr/include/stdio.h
779 file path=usr/include/stdio_ext.h
780 file path=usr/include/stdio_impl.h
781 file path=usr/include/stdio_tag.h
782 file path=usr/include/stdlib.h
783 file path=usr/include/storclass.h
784 file path=usr/include/string.h
785 file path=usr/include/strings.h
786 file path=usr/include/stropts.h
787 file path=usr/include/syms.h
```

```
file path=usr/include/synch.h
file path=usr/include/sys/acct.h
file path=usr/include/sys/acctctl.h
file path=usr/include/sys/acl.h
file path=usr/include/sys/acl_impl.h
file path=usr/include/sys/acpi_drv.h
file path=usr/include/sys/aio.h
file path=usr/include/sys/aio_impl.h
file path=usr/include/sys/aio_req.h
file path=usr/include/sys/aiocb.h
file path=usr/include/sys/archsystm.h
file path=usr/include/sys/ascii.h
file path=usr/include/sys/asm_linkage.h
file path=usr/include/sys/asynch.h
file path=usr/include/sys/atomic.h
file path=usr/include/sys/attr.h
file path=usr/include/sys/autoconf.h
file path=usr/include/sys/auxv.h
file path=usr/include/sys/auxv_386.h
file path=usr/include/sys/auxv_SPARC.h
file path=usr/include/sys/av/iec61883.h
file path=usr/include/sys/avintr.h
file path=usr/include/sys/avl.h
file path=usr/include/sys/avl_impl.h
file path=usr/include/sys/bitmap.h
file path=usr/include/sys/bitset.h
file path=usr/include/sys/bl.h
file path=usr/include/sys/blkdev.h
file path=usr/include/sys/bmc_intf.h
file path=usr/include/sys/bofi.h
file path=usr/include/sys/bofi_impl.h
file path=usr/include/sys/bootconf.h
$(i386_ONLY)file path=usr/include/sys/bootregs.h
file path=usr/include/sys/bootstat.h
$(i386_ONLY)file path=usr/include/sys/bootsvcs.h
file path=usr/include/sys/bpp_io.h
file path=usr/include/sys/brand.h
file path=usr/include/sys/buf.h
file path=usr/include/sys/bufmod.h
file path=usr/include/sys/bustypes.h
file path=usr/include/sys/byteorder.h
file path=usr/include/sys/callb.h
file path=usr/include/sys/callo.h
file path=usr/include/sys/cap_util.h
file path=usr/include/sys/ccompile.h
file path=usr/include/sys/cdio.h
file path=usr/include/sys/cis.h
file path=usr/include/sys/cis_handlers.h
file path=usr/include/sys/cis_protos.h
file path=usr/include/sys/cladm.h
file path=usr/include/sys/class.h
file path=usr/include/sys/clconf.h
file path=usr/include/sys/cmlb.h
file path=usr/include/sys/cmn_err.h
$(sparc_ONLY)file path=usr/include/sys/cmpregs.h
file path=usr/include/sys/compress.h
file path=usr/include/sys/condvar.h
file path=usr/include/sys/condvar_impl.h
file path=usr/include/sys/conf.h
file path=usr/include/sys/consdev.h
file path=usr/include/sys/console.h
file path=usr/include/sys/consplat.h
file path=usr/include/sys/contract.h
file path=usr/include/sys/contract/device.h
file path=usr/include/sys/contract/device_impl.h
file path=usr/include/sys/contract/process.h
```



```
file path=usr/include/sys/contract/process_impl.h
file path=usr/include/sys/contract_impl.h
$(i386_ONLY)file path=usr/include/sys/controlregs.h
file path=usr/include/sys/copyops.h
file path=usr/include/sys/core.h
file path=usr/include/sys/corectl.h
file path=usr/include/sys/cpc_impl.h
file path=usr/include/sys/cpc_pcbe.h
file path=usr/include/sys/cpr.h
file path=usr/include/sys/cpu.h
file path=usr/include/sys/cpucaps.h
file path=usr/include/sys/cpucaps_impl.h
file path=usr/include/sys/cpupart.h
file path=usr/include/sys/cpuvar.h
file path=usr/include/sys/crc32.h
file path=usr/include/sys/cred.h
file path=usr/include/sys/cred_impl.h
file path=usr/include/sys/crtctl.h
file path=usr/include/sys/crypto/api.h
file path=usr/include/sys/crypto/common.h
file path=usr/include/sys/crypto/ioctl.h
file path=usr/include/sys/crypto/ioctladmin.h
file path=usr/include/sys/crypto/spi.h
file path=usr/include/sys/cs.h
file path=usr/include/sys/cs_priv.h
file path=usr/include/sys/cs_strings.h
file path=usr/include/sys/cs_stubs.h
file path=usr/include/sys/cs_types.h
file path=usr/include/sys/csiioctl.h
file path=usr/include/sys/ctf.h
file path=usr/include/sys/ctf_api.h
file path=usr/include/sys/ctfs.h
file path=usr/include/sys/ctfs_impl.h
file path=usr/include/sys/ctype.h
file path=usr/include/sys/cyclic.h
file path=usr/include/sys/cyclic_impl.h
file path=usr/include/sys/dacf.h
file path=usr/include/sys/dacf_impl.h
file path=usr/include/sys/damap.h
file path=usr/include/sys/damap_impl.h
file path=usr/include/sys/dc_ki.h
file path=usr/include/sys/ddi.h
file path=usr/include/sys/ddi_hp.h
file path=usr/include/sys/ddi_hp_impl.h
file path=usr/include/sys/ddi_impldefs.h
file path=usr/include/sys/ddi_implfuncs.h
file path=usr/include/sys/ddi_intr.h
file path=usr/include/sys/ddi_intr_impl.h
file path=usr/include/sys/ddi_isa.h
file path=usr/include/sys/ddi_obsolete.h
file path=usr/include/sys/ddi_timer.h
file path=usr/include/sys/ddidevmap.h
file path=usr/include/sys/ddidmareq.h
file path=usr/include/sys/ddifm.h
file path=usr/include/sys/ddifm_impl.h
file path=usr/include/sys/ddimapreq.h
file path=usr/include/sys/ddipropdefs.h
file path=usr/include/sys/dditypes.h
file path=usr/include/sys/debug.h
$(i386_ONLY)file path=usr/include/sys/debugreg.h
file path=usr/include/sys/des.h
file path=usr/include/sys/devcache.h
file path=usr/include/sys/devcache_impl.h
file path=usr/include/sys/devctl.h
file path=usr/include/sys/devfm.h
file path=usr/include/sys/devid_cache.h
```

```
file path=usr/include/sys/devinfo_impl.h
file path=usr/include/sys/devops.h
file path=usr/include/sys/devpolicy.h
file path=usr/include/sys/devpoll.h
file path=usr/include/sys/dirent.h
file path=usr/include/sys/disp.h
file path=usr/include/sys/dkbad.h
file path=usr/include/sys/dkio.h
file path=usr/include/sys/dklabel.h
$(sparc_ONLY)file path=usr/include/sys/dkmpio.h
$(i386_ONLY)file path=usr/include/sys/dktp/altsctr.h
$(i386_ONLY)file path=usr/include/sys/dktp/cmpkt.h
file path=usr/include/sys/dktp/dadkio.h
file path=usr/include/sys/dktp/fdisk.h
file path=usr/include/sys/dl.h
file path=usr/include/sys/dld.h
file path=usr/include/sys/dlpi.h
file path=usr/include/sys/dls_mgmt.h
$(i386_ONLY)file path=usr/include/sys/dma_engine.h
file path=usr/include/sys/dma_i8237A.h
file path=usr/include/sys/dnlc.h
file path=usr/include/sys/door.h
file path=usr/include/sys/door_data.h
file path=usr/include/sys/door_impl.h
file path=usr/include/sys/dumphdr.h
file path=usr/include/sys/ecppio.h
file path=usr/include/sys/ecppreg.h
file path=usr/include/sys/ecppsys.h
file path=usr/include/sys/ecppvar.h
file path=usr/include/sys/efi_partition.h
file path=usr/include/sys/elf.h
file path=usr/include/sys/elf_386.h
file path=usr/include/sys/elf_SPARC.h
file path=usr/include/sys/elf_amd64.h
file path=usr/include/sys/elf_notes.h
file path=usr/include/sys/elftypes.h
file path=usr/include/sys/epm.h
file path=usr/include/sys/errno.h
file path=usr/include/sys/errorq.h
file path=usr/include/sys/errorq_impl.h
file path=usr/include/sys/esunddi.h
file path=usr/include/sys/ethernet.h
file path=usr/include/sys/euc.h
file path=usr/include/sys/eucioctl.h
file path=usr/include/sys/exacct.h
file path=usr/include/sys/exacct_catalog.h
file path=usr/include/sys/exacct_impl.h
file path=usr/include/sys/exec.h
file path=usr/include/sys/exechdr.h
file path=usr/include/sys/fault.h
file path=usr/include/sys/fbio.h
file path=usr/include/sys/fbuf.h
file path=usr/include/sys/fc4/fc.h
file path=usr/include/sys/fc4/fc_transport.h
file path=usr/include/sys/fc4/fcal.h
file path=usr/include/sys/fc4/fcal_linkapp.h
file path=usr/include/sys/fc4/fcal_transport.h
file path=usr/include/sys/fc4/fcio.h
file path=usr/include/sys/fc4/fcp.h
file path=usr/include/sys/fc4/linkapp.h
file path=usr/include/sys/fcntl.h
file path=usr/include/sys/fdbuffer.h
file path=usr/include/sys/fdio.h
$(sparc_ONLY)file path=usr/include/sys/fdreg.h
$(sparc_ONLY)file path=usr/include/sys/fdvar.h
file path=usr/include/sys/feature_tests.h
```



```
file path=usr/include/sys/fem.h
file path=usr/include/sys/file.h
file path=usr/include/sys/filio.h
file path=usr/include/sys/flock.h
file path=usr/include/sys/flock_impl.h
$(sparc_ONLY)file path=usr/include/sys/fm/cpu/SPARC64-VI.h
$(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-II.h
$(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-III.h
$(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-T1.h
file path=usr/include/sys/fm/fs/zfs.h
file path=usr/include/sys/fm/io/ddi.h
file path=usr/include/sys/fm/io/disk.h
file path=usr/include/sys/fm/io/opl_mc_fm.h
file path=usr/include/sys/fm/io/pci.h
file path=usr/include/sys/fm/io/scsi.h
file path=usr/include/sys/fm/io/sun4upci.h
file path=usr/include/sys/fm/protocol.h
file path=usr/include/sys/fm/util.h
file path=usr/include/sys/fork.h
$(i386_ONLY)file path=usr/include/sys/fp.h
$(sparc_ONLY)file path=usr/include/sys/fpu/fpu_simulator.h
$(sparc_ONLY)file path=usr/include/sys/fpu/fpusystm.h
$(sparc_ONLY)file path=usr/include/sys/fpu/globals.h
$(sparc_ONLY)file path=usr/include/sys/fpu/ieee.h
file path=usr/include/sys/frame.h
file path=usr/include/sys/fs/autofs.h
file path=usr/include/sys/fs/cachefs_dir.h
file path=usr/include/sys/fs/cachefs_dlog.h
file path=usr/include/sys/fs/cachefs_filegrp.h
file path=usr/include/sys/fs/cachefs_fs.h
file path=usr/include/sys/fs/cachefs_fscache.h
file path=usr/include/sys/fs/cachefs_ioctl.h
file path=usr/include/sys/fs/cachefs_log.h
file path=usr/include/sys/fs/decomp.h
file path=usr/include/sys/fs/dv_node.h
file path=usr/include/sys/fs/fifonode.h
file path=usr/include/sys/fs/hsfs_isospec.h
file path=usr/include/sys/fs/hsfs_node.h
file path=usr/include/sys/fs/hsfs_rrip.h
file path=usr/include/sys/fs/hsfs_spec.h
file path=usr/include/sys/fs/hsfs_susp.h
file path=usr/include/sys/fs/lofs_info.h
file path=usr/include/sys/fs/lofs_node.h
file path=usr/include/sys/fs/mntdata.h
file path=usr/include/sys/fs/namenode.h
file path=usr/include/sys/fs/pc_dir.h
file path=usr/include/sys/fs/pc_fs.h
file path=usr/include/sys/fs/pc_label.h
file path=usr/include/sys/fs/pc_node.h
file path=usr/include/sys/fs/pxfs_ki.h
file path=usr/include/sys/fs/sdev_impl.h
file path=usr/include/sys/fs/snode.h
file path=usr/include/sys/fs/swapnode.h
file path=usr/include/sys/fs/tmp.h
file path=usr/include/sys/fs/tmpnode.h
file path=usr/include/sys/fs/udf_inode.h
file path=usr/include/sys/fs/udf_volume.h
file path=usr/include/sys/fs/ufs_acl.h
file path=usr/include/sys/fs/ufs_bio.h
file path=usr/include/sys/fs/ufs_filio.h
file path=usr/include/sys/fs/ufs_fs.h
file path=usr/include/sys/fs/ufs_fsdir.h
file path=usr/include/sys/fs/ufs_inode.h
file path=usr/include/sys/fs/ufs_lockfs.h
file path=usr/include/sys/fs/ufs_log.h
file path=usr/include/sys/fs/ufs_mount.h
```

```
1052 file path=usr/include/sys/fs/ufs_panic.h
1053 file path=usr/include/sys/fs/ufs_prot.h
1054 file path=usr/include/sys/fs/ufs_quota.h
1055 file path=usr/include/sys/fs/ufs_snap.h
1056 file path=usr/include/sys/fs/ufs_trans.h
1057 file path=usr/include/sys/fs/zfs.h
1058 file path=usr/include/sys/fs_reparse.h
1059 file path=usr/include/sys/fs_subr.h
1060 file path=usr/include/sys/fsid.h
1061 $(sparc_ONLY)file path=usr/include/sys/fsr.h
1062 file path=usr/include/sys/fss.h
1063 file path=usr/include/sys/fssnap.h
1064 file path=usr/include/sys/fssnap_if.h
1065 file path=usr/include/sys/fsspriocntl.h
1066 file path=usr/include/sys/fstyp.h
1067 file path=usr/include/sys/ftrace.h
1068 file path=usr/include/sys/fx.h
1069 file path=usr/include/sys/fxpriocntl.h
1070 file path=usr/include/sys/gfs.h
1071 file path=usr/include/sys/gld.h
1072 file path=usr/include/sys/gldpriv.h
1073 file path=usr/include/sys/group.h
1074 file path=usr/include/sys/hdio.h
1075 file path=usr/include/sys/hook.h
1076 file path=usr/include/sys/hook_event.h
1077 file path=usr/include/sys/hook_impl.h
1078 file path=usr/include/sys/hotplug/hpcsvc.h
1079 file path=usr/include/sys/hotplug/hpctrl.h
1080 file path=usr/include/sys/hotplug/pci/pcicfg.h
1081 file path=usr/include/sys/hotplug/pci/pcihp.h
1082 file path=usr/include/sys/hwconf.h
1083 $(i386_ONLY)file path=usr/include/sys/hypervisor.h
1084 $(i386_ONLY)file path=usr/include/sys/i8272A.h
1085 file path=usr/include/sys/ia.h
1086 file path=usr/include/sys/iapriocntl.h
1087 file path=usr/include/sys/ib/adapters/hermon/hermon_ioctl.h
1088 file path=usr/include/sys/ib/adapters/mlnx_umap.h
1089 file path=usr/include/sys/ib/adapters/tavor/tavor_ioctl.h
1090 file path=usr/include/sys/ib/clients/ibd/ibd.h
1091 file path=usr/include/sys/ib/clients/of/ofa_solaris.h
1092 file path=usr/include/sys/ib/clients/of/ofed_kernel.h
1093 file path=usr/include/sys/ib/clients/of/rdma/ib_addr.h
1094 file path=usr/include/sys/ib/clients/of/rdma/ib_user_mad.h
1095 file path=usr/include/sys/ib/clients/of/rdma/ib_user_sa.h
1096 file path=usr/include/sys/ib/clients/of/rdma/ib_user_verbs.h
1097 file path=usr/include/sys/ib/clients/of/rdma/ib_verbs.h
1098 file path=usr/include/sys/ib/clients/of/rdma/rdma_cm.h
1099 file path=usr/include/sys/ib/clients/of/rdma/rdma_user_cm.h
1100 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_cma.h
1101 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ib_cma.h
1102 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_kverb_impl.h
1103 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ofs_common.h
1104 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_rdma_user_cm.h
1105 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_ucma.h
1106 file path=usr/include/sys/ib/clients/of/sol_umad/sol_umad.h
1107 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs.h
1108 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs2ucma.h
1109 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_comp.h
1110 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_event.h
1111 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_hca.h
1112 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_qp.h
1113 file path=usr/include/sys/ib/ib_pkt_hdrs.h
1114 file path=usr/include/sys/ib/ib_types.h
1115 file path=usr/include/sys/ib/ibnex/ibnex_devctl.h
1116 file path=usr/include/sys/ib/ibtl/ibci.h
1117 file path=usr/include/sys/ib/ibtl/ibti.h
```



```
1118 file path=usr/include/sys/ib/ibtl/ibti_cm.h
1119 file path=usr/include/sys/ib/ibtl/ibti_common.h
1120 file path=usr/include/sys/ib/ibtl/ibtl_ci_types.h
1121 file path=usr/include/sys/ib/ibtl/ibtl_status.h
1122 file path=usr/include/sys/ib/ibtl/ibtl_types.h
1123 file path=usr/include/sys/ib/ibtl/ibvti.h
1124 file path=usr/include/sys/ib/ibtl/impl/ibtl_util.h
1125 file path=usr/include/sys/ib/mgt/ib_dm_attr.h
1126 file path=usr/include/sys/ib/mgt/ib_mad.h
1127 file path=usr/include/sys/ib/mgt/ibmf/ibmf.h
1128 file path=usr/include/sys/ib/mgt/ibmf/ibmf_msg.h
1129 file path=usr/include/sys/ib/mgt/ibmf/ibmf_saa.h
1130 file path=usr/include/sys/ib/mgt/ibmf/ibmf_utils.h
1131 file path=usr/include/sys/ib/mgt/sa_recs.h
1132 file path=usr/include/sys/ib/mgt/sm_attr.h
1133 file path=usr/include/sys/ibpart.h
1134 file path=usr/include/sys/id32.h
1135 file path=usr/include/sys/id_space.h
1136 file path=usr/include/sys/idmap.h
1137 file path=usr/include/sys/inline.h
1138 file path=usr/include/sys/instance.h
1139 file path=usr/include/sys/int_const.h
1140 file path=usr/include/sys/int_fmtio.h
1141 file path=usr/include/sys/int_limits.h
1142 file path=usr/include/sys/int_types.h
1143 file path=usr/include/sys/inttypes.h
1144 file path=usr/include/sys/ioccom.h
1145 file path=usr/include/sys/ioctl.h
1146 $(i386_ONLY)file path=usr/include/sys/iommulib.h
1147 file path=usr/include/sys/ipc.h
1148 file path=usr/include/sys/ipc_impl.h
1149 file path=usr/include/sys/ipc_rctl.h
1150 file path=usr/include/sys/isa_defs.h
1151 file path=usr/include/sys/iso/signal_iso.h
1152 file path=usr/include/sys/jioctl.h
1153 file path=usr/include/sys/kbd.h
1154 file path=usr/include/sys/kbdreg.h
1155 file path=usr/include/sys/kbio.h
1156 file path=usr/include/sys/kcpc.h
1157 file path=usr/include/sys/kd.h
1158 file path=usr/include/sys/kdi.h
1159 file path=usr/include/sys/kdi_impl.h
1160 file path=usr/include/sys/kdi_machimpl.h
1161 $(i386_ONLY)file path=usr/include/sys/kdi_regs.h
1162 file path=usr/include/sys/kiconv.h
1163 file path=usr/include/sys/kidmap.h
1164 file path=usr/include/sys/klpd.h
1165 file path=usr/include/sys/klwp.h
1166 file path=usr/include/sys/kmem.h
1167 file path=usr/include/sys/kmem_impl.h
1168 file path=usr/include/sys/kobj.h
1169 file path=usr/include/sys/kobj_impl.h
1170 file path=usr/include/sys/ksocket.h
1171 file path=usr/include/sys/kstat.h
1172 file path=usr/include/sys/kstr.h
1173 file path=usr/include/sys/ksyms.h
1174 file path=usr/include/sys/ksynch.h
1175 file path=usr/include/sys/lc_core.h
1176 file path=usr/include/sys/ldterm.h
1177 file path=usr/include/sys/lgrp.h
1178 file path=usr/include/sys/lgrp_user.h
1179 file path=usr/include/sys/link.h
1180 file path=usr/include/sys/list.h
1181 file path=usr/include/sys/list_impl.h
1182 file path=usr/include/sys/llc1.h
1183 file path=usr/include/sys/loadavg.h
```

```
1184 file path=usr/include/sys/localedef.h
1185 file path=usr/include/sys/lock.h
1186 file path=usr/include/sys/lockfs.h
1187 file path=usr/include/sys/lofi.h
1188 file path=usr/include/sys/log.h
1189 file path=usr/include/sys/logindmux.h
1190 file path=usr/include/sys/lvm/md_basic.h
1191 file path=usr/include/sys/lvm/md_convert.h
1192 file path=usr/include/sys/lvm/md_crc.h
1193 file path=usr/include/sys/lvm/md_hotspares.h
1194 file path=usr/include/sys/lvm/md_mddb.h
1195 file path=usr/include/sys/lvm/md_mdiox.h
1196 file path=usr/include/sys/lvm/md_mhdx.h
1197 file path=usr/include/sys/lvm/md_mirror.h
1198 file path=usr/include/sys/lvm/md_mirror_shared.h
1199 file path=usr/include/sys/lvm/md_names.h
1200 file path=usr/include/sys/lvm/md_notify.h
1201 file path=usr/include/sys/lvm/md_raid.h
1202 file path=usr/include/sys/lvm/md_rename.h
1203 file path=usr/include/sys/lvm/md_sp.h
1204 file path=usr/include/sys/lvm/md_stripe.h
1205 file path=usr/include/sys/lvm/md_trans.h
1206 file path=usr/include/sys/lvm/mdio.h
1207 file path=usr/include/sys/lvm/mdmed.h
1208 file path=usr/include/sys/lvm/mdmn_commd.h
1209 file path=usr/include/sys/lvm/mdvar.h
1210 file path=usr/include/sys/lwp.h
1211 file path=usr/include/sys/lwp_timer_impl.h
1212 file path=usr/include/sys/lwp_upimutex_impl.h
1213 file path=usr/include/sys/mac.h
1214 file path=usr/include/sys/mac_ether.h
1215 file path=usr/include/sys/mac_flow.h
1216 file path=usr/include/sys/mac_provider.h
1217 file path=usr/include/sys/machelf.h
1218 file path=usr/include/sys/machlock.h
1219 file path=usr/include/sys/machsig.h
1220 file path=usr/include/sys/machtypes.h
1221 file path=usr/include/sys/map.h
1222 $(i386_ONLY)file path=usr/include/sys/mc.h
1223 $(i386_ONLY)file path=usr/include/sys/mc_amd.h
1224 $(i386_ONLY)file path=usr/include/sys/mc_intel.h
1225 $(i386_ONLY)file path=usr/include/sys/mca_amd.h
1226 $(i386_ONLY)file path=usr/include/sys/mca_x86.h
1227 file path=usr/include/sys/md4.h
1228 file path=usr/include/sys/md5.h
1229 file path=usr/include/sys/md5_consts.h
1230 file path=usr/include/sys/mdi_impldefs.h
1231 file path=usr/include/sys/mem.h
1232 file path=usr/include/sys/mem_config.h
1233 file path=usr/include/sys/memlist.h
1234 file path=usr/include/sys/mhd.h
1235 file path=usr/include/sys/mii.h
1236 file path=usr/include/sys/miiregs.h
1237 file path=usr/include/sys/mkdev.h
1238 file path=usr/include/sys/mman.h
1239 file path=usr/include/sys/mmapobj.h
1240 file path=usr/include/sys/mntent.h
1241 file path=usr/include/sys/mntio.h
1242 file path=usr/include/sys/mnttab.h
1243 file path=usr/include/sys/modctl.h
1244 file path=usr/include/sys/mode.h
1245 file path=usr/include/sys/model.h
1246 file path=usr/include/sys/modhash.h
1247 file path=usr/include/sys/modhash_impl.h
1248 file path=usr/include/sys/mount.h
1249 file path=usr/include/sys/mouse.h
```



```
1250 file path=usr/include/sys/msacct.h
1251 file path=usr/include/sys/msg.h
1252 file path=usr/include/sys/msg_impl.h
1253 file path=usr/include/sys/msio.h
1254 file path=usr/include/sys/msreg.h
1255 file path=usr/include/sys/mtio.h
1256 file path=usr/include/sys/multidata.h
1257 file path=usr/include/sys/mutex.h
1258 $(i386_ONLY)file path=usr/include/sys/mutex_impl.h
1259 file path=usr/include/sys/nbmlock.h
1260 file path=usr/include/sys/ndi_impldefs.h
1261 file path=usr/include/sys/ndifm.h
1262 file path=usr/include/sys/netconfig.h
1263 file path=usr/include/sys/neti.h
1264 file path=usr/include/sys/netstack.h
1265 file path=usr/include/sys/nexusdefs.h
1266 file path=usr/include/sys/note.h
1267 file path=usr/include/sys/nvpair.h
1268 file path=usr/include/sys/nvpair_impl.h
1269 file path=usr/include/sys/objfs.h
1270 file path=usr/include/sys/objfs_impl.h
1271 file path=usr/include/sys/obpdefs.h
1272 file path=usr/include/sys/old_procfs.h
1273 file path=usr/include/sys/open.h
1274 file path=usr/include/sys/openpromio.h
1275 file path=usr/include/sys/panic.h
1276 file path=usr/include/sys/param.h
1277 file path=usr/include/sys/pathconf.h
1278 file path=usr/include/sys/pathname.h
1279 file path=usr/include/sys/pattr.h
1280 file path=usr/include/sys/pbio.h
1281 file path=usr/include/sys/pcb.h
1282 file path=usr/include/sys/pccard.h
1283 file path=usr/include/sys/pci.h
1284 $(i386_ONLY)file path=usr/include/sys/pcic_reg.h
1285 $(i386_ONLY)file path=usr/include/sys/pcic_var.h
1286 file path=usr/include/sys/pcie.h
1287 file path=usr/include/sys/pcmcia.h
1288 file path=usr/include/sys/pcmcia/pcata.h
1289 file path=usr/include/sys/pcmcia/pcser_conf.h
1290 file path=usr/include/sys/pcmcia/pcser_io.h
1291 file path=usr/include/sys/pcmcia/pcser_manuspec.h
1292 file path=usr/include/sys/pcmcia/pcser_reg.h
1293 file path=usr/include/sys/pcmcia/pcser_var.h
1294 file path=usr/include/sys/pctypes.h
1295 file path=usr/include/sys/pfmod.h
1296 file path=usr/include/sys/pg.h
1297 file path=usr/include/sys/pghw.h
1298 file path=usr/include/sys/physmem.h
1299 $(i386_ONLY)file path=usr/include/sys/pic.h
1300 $(i386_ONLY)file path=usr/include/sys/pit.h
1301 file path=usr/include/sys/pkp_hash.h
1302 file path=usr/include/sys/pm.h
1303 $(i386_ONLY)file path=usr/include/sys/pmem.h
1304 file path=usr/include/sys/policy.h
1305 file path=usr/include/sys/poll.h
1306 file path=usr/include/sys/poll_impl.h
1307 file path=usr/include/sys/pool.h
1308 file path=usr/include/sys/pool_impl.h
1309 file path=usr/include/sys/pool_pset.h
1310 file path=usr/include/sys/port.h
1311 file path=usr/include/sys/port_impl.h
1312 file path=usr/include/sys/port_kernel.h
1313 file path=usr/include/sys/ppmio.h
1314 file path=usr/include/sys/priocntl.h
1315 file path=usr/include/sys/priv.h
```

```
1316 file path=usr/include/sys/priv_const.h
1317 file path=usr/include/sys/priv_impl.h
1318 file path=usr/include/sys/priv_names.h
1319 $(i386_ONLY)file path=usr/include/sys/privmregs.h
1320 $(i386_ONLY)file path=usr/include/sys/privregs.h
1321 file path=usr/include/sys/prnio.h
1322 file path=usr/include/sys/proc.h
1323 file path=usr/include/sys/proc/prdata.h
1324 file path=usr/include/sys/processor.h
1325 file path=usr/include/sys/procfs.h
1326 file path=usr/include/sys/procfs_isa.h
1327 file path=usr/include/sys/procset.h
1328 file path=usr/include/sys/project.h
1329 $(i386_ONLY)file path=usr/include/sys/prom_emul.h
1330 $(i386_ONLY)file path=usr/include/sys/prom_isa.h
1331 $(i386_ONLY)file path=usr/include/sys/prom_plat.h
1332 file path=usr/include/sys/promif.h
1333 file path=usr/include/sys/promimpl.h
1334 file path=usr/include/sys/protosw.h
1335 file path=usr/include/sys/prsystm.h
1336 file path=usr/include/sys/pset.h
1337 file path=usr/include/sys/psw.h
1338 $(i386_ONLY)file path=usr/include/sys/pte.h
1339 file path=usr/include/sys/ptem.h
1340 file path=usr/include/sys/ptms.h
1341 file path=usr/include/sys/ptyvar.h
1342 file path=usr/include/sys/queue.h
1343 file path=usr/include/sys/raidioctl.h
1344 file path=usr/include/sys/ramdisk.h
1345 file path=usr/include/sys/random.h
1346 file path=usr/include/sys/rctl.h
1347 file path=usr/include/sys/rctl_impl.h
1348 file path=usr/include/sys/rds.h
1349 file path=usr/include/sys/reboot.h
1350 file path=usr/include/sys/refstr.h
1351 file path=usr/include/sys/refstr_impl.h
1352 file path=usr/include/sys/reg.h
1353 file path=usr/include/sys/regset.h
1354 file path=usr/include/sys/resource.h
1355 file path=usr/include/sys/rlioctl.h
1356 file path=usr/include/sys/rsm/rsm.h
1357 file path=usr/include/sys/rsm/rsm_common.h
1358 file path=usr/include/sys/rsm/rsmapi_common.h
1359 file path=usr/include/sys/rsm/rsmka_path_int.h
1360 file path=usr/include/sys/rsm/rsmndi.h
1361 file path=usr/include/sys/rsm/rsmpi.h
1362 file path=usr/include/sys/rsm/rsmpi_driver.h
1363 file path=usr/include/sys/rt.h
1364 $(i386_ONLY)file path=usr/include/sys/rtc.h
1365 file path=usr/include/sys/rtpriocntl.h
1366 file path=usr/include/sys/rwlock.h
1367 file path=usr/include/sys/rwlock_impl.h
1368 file path=usr/include/sys/rwstlock.h
1369 file path=usr/include/sys/sad.h
1370 $(i386_ONLY)file path=usr/include/sys/sata/sata_defs.h
1371 $(i386_ONLY)file path=usr/include/sys/sata/sata_hba.h
1372 file path=usr/include/sys/schedctl.h
1373 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/ifpio.h
1374 file path=usr/include/sys/scsi/adapters/scsi_vhci.h
1375 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/sfvar.h
1376 file path=usr/include/sys/scsi/conf/autoconf.h
1377 file path=usr/include/sys/scsi/conf/device.h
1378 file path=usr/include/sys/scsi/generic/commands.h
1379 file path=usr/include/sys/scsi/generic/dad_mode.h
1380 file path=usr/include/sys/scsi/generic/inquiry.h
1381 file path=usr/include/sys/scsi/generic/message.h
```



```
1382 file path=usr/include/sys/scsi/generic/mode.h
1383 file path=usr/include/sys/scsi/generic/persist.h
1384 file path=usr/include/sys/scsi/generic/sense.h
1385 file path=usr/include/sys/scsi/generic/sff_frames.h
1386 file path=usr/include/sys/scsi/generic/smp_frames.h
1387 file path=usr/include/sys/scsi/generic/status.h
1388 file path=usr/include/sys/scsi/impl/commands.h
1389 file path=usr/include/sys/scsi/impl/inquiry.h
1390 file path=usr/include/sys/scsi/impl/mode.h
1391 file path=usr/include/sys/scsi/impl/scsi_reset_notify.h
1392 file path=usr/include/sys/scsi/impl/scsi_sas.h
1393 file path=usr/include/sys/scsi/impl/sense.h
1394 file path=usr/include/sys/scsi/impl/services.h
1395 file path=usr/include/sys/scsi/impl/smp_transport.h
1396 file path=usr/include/sys/scsi/impl/spc3_types.h
1397 file path=usr/include/sys/scsi/impl/status.h
1398 file path=usr/include/sys/scsi/impl/transport.h
1399 file path=usr/include/sys/scsi/impl/types.h
1400 file path=usr/include/sys/scsi/impl/uscsi.h
1401 file path=usr/include/sys/scsi/impl/usmp.h
1402 file path=usr/include/sys/scsi/scsi.h
1403 file path=usr/include/sys/scsi/scsi_address.h
1404 file path=usr/include/sys/scsi/scsi_ctl.h
1405 file path=usr/include/sys/scsi/scsi_fm.h
1406 file path=usr/include/sys/scsi/scsi_params.h
1407 file path=usr/include/sys/scsi/scsi_pkt.h
1408 file path=usr/include/sys/scsi/scsi_resource.h
1409 file path=usr/include/sys/scsi/scsi_types.h
1410 file path=usr/include/sys/scsi/scsi_watch.h
1411 file path=usr/include/sys/scsi/targets/sddef.h
1412 file path=usr/include/sys/scsi/targets/ses.h
1413 file path=usr/include/sys/scsi/targets/sesio.h
1414 file path=usr/include/sys/scsi/targets/sgendef.h
1415 file path=usr/include/sys/scsi/targets/smp.h
1416 $(sparc_ONLY)file path=usr/include/sys/scsi/targets/ssddef.h
1417 file path=usr/include/sys/scsi/targets/stdef.h
1418 $(i386_ONLY)file path=usr/include/sys/segment.h
1419 $(i386_ONLY)file path=usr/include/sys/segments.h
1420 file path=usr/include/sys/select.h
1421 file path=usr/include/sys/sem.h
1422 file path=usr/include/sys/sem_impl.h
1423 file path=usr/include/sys/sema_impl.h
1424 file path=usr/include/sys/semaphore.h
1425 file path=usr/include/sys/sendfile.h
1426 $(sparc_ONLY)file path=usr/include/sys/ser_async.h
1427 file path=usr/include/sys/ser_sync.h
1428 $(sparc_ONLY)file path=usr/include/sys/ser_zscc.h
1429 file path=usr/include/sys/serializer.h
1430 file path=usr/include/sys/session.h
1431 file path=usr/include/sys/sha1.h
1432 file path=usr/include/sys/sha2.h
1433 file path=usr/include/sys/share.h
1434 file path=usr/include/sys/shm.h
1435 file path=usr/include/sys/shm_impl.h
1436 file path=usr/include/sys/sid.h
1437 file path=usr/include/sys/siginfo.h
1438 file path=usr/include/sys/signal.h
1439 file path=usr/include/sys/sleepq.h
1440 file path=usr/include/sys/smbios.h
1441 file path=usr/include/sys/smbios_impl.h
1442 file path=usr/include/sys/smedia.h
1443 file path=usr/include/sys/sobject.h
1444 $(sparc_ONLY)file path=usr/include/sys/socal_cq_defs.h
1445 $(sparc_ONLY)file path=usr/include/sys/socalio.h
1446 $(sparc_ONLY)file path=usr/include/sys/socalmap.h
1447 $(sparc_ONLY)file path=usr/include/sys/socalreg.h
```

```
$(sparc_ONLY)file path=usr/include/sys/socalvar.h
file path=usr/include/sys/socket.h
file path=usr/include/sys/socket_impl.h
file path=usr/include/sys/socket_proto.h
file path=usr/include/sys/socketvar.h
file path=usr/include/sys/sockio.h
file path=usr/include/sys/spl.h
file path=usr/include/sys/squeue.h
file path=usr/include/sys/squeue_impl.h
file path=usr/include/sys/sservice.h
file path=usr/include/sys/stack.h
file path=usr/include/sys/stat.h
file path=usr/include/sys/stat_impl.h
file path=usr/include/sys/statfs.h
file path=usr/include/sys/statvfs.h
file path=usr/include/sys/stdbool.h
file path=usr/include/sys/stdint.h
file path=usr/include/sys/stermio.h
file path=usr/include/sys/stream.h
file path=usr/include/sys/strft.h
file path=usr/include/sys/strlog.h
file path=usr/include/sys/strmdep.h
file path=usr/include/sys/stropts.h
file path=usr/include/sys/strredir.h
file path=usr/include/sys/strstat.h
file path=usr/include/sys/strsubr.h
file path=usr/include/sys/strsun.h
file path=usr/include/sys/strtty.h
file path=usr/include/sys/sunddi.h
file path=usr/include/sys/sunldi.h
file path=usr/include/sys/sunldi_impl.h
file path=usr/include/sys/sunmdi.h
file path=usr/include/sys/sunndi.h
file path=usr/include/sys/sunpm.h
file path=usr/include/sys/suntpi.h
file path=usr/include/sys/suntty.h
file path=usr/include/sys/swap.h
file path=usr/include/sys/synch.h
file path=usr/include/sys/syscall.h
file path=usr/include/sys/sysconf.h
file path=usr/include/sys/sysconfig.h
file path=usr/include/sys/sysconfig_impl.h
file path=usr/include/sys/sysdc.h
file path=usr/include/sys/sysdc_impl.h
file path=usr/include/sys/sysevent.h
file path=usr/include/sys/sysevent/ap_driver.h
file path=usr/include/sys/sysevent/dev.h
file path=usr/include/sys/sysevent/domain.h
file path=usr/include/sys/sysevent/dr.h
file path=usr/include/sys/sysevent/env.h
file path=usr/include/sys/sysevent/eventdefs.h
file path=usr/include/sys/sysevent/ipmp.h
file path=usr/include/sys/sysevent/pwrctl.h
file path=usr/include/sys/sysevent/svm.h
file path=usr/include/sys/sysevent/vrrp.h
file path=usr/include/sys/sysevent_impl.h
$(i386_ONLY)file path=usr/include/sys/sysi86.h
file path=usr/include/sys/sysinfo.h
file path=usr/include/sys/syslog.h
file path=usr/include/sys/sysmacros.h
file path=usr/include/sys/systeminfo.h
file path=usr/include/sys/systm.h
file path=usr/include/sys/t_kuser.h
file path=usr/include/sys/t_lock.h
file path=usr/include/sys/task.h
file path=usr/include/sys/taskq.h
```



```
file path=usr/include/sys/taskq_impl.h
file path=usr/include/sys/telioctl.h
file path=usr/include/sys/termio.h
file path=usr/include/sys/termios.h
file path=usr/include/sys/termiox.h
file path=usr/include/sys/thread.h
file path=usr/include/sys/ticlts.h
file path=usr/include/sys/ticots.h
file path=usr/include/sys/ticotsord.h
file path=usr/include/sys/tihdr.h
file path=usr/include/sys/time.h
file path=usr/include/sys/time_impl.h
file path=usr/include/sys/time_std_impl.h
file path=usr/include/sys/timeb.h
file path=usr/include/sys/timer.h
file path=usr/include/sys/times.h
file path=usr/include/sys/timex.h
file path=usr/include/sys/timod.h
file path=usr/include/sys/tirdwr.h
file path=usr/include/sys/tiuser.h
file path=usr/include/sys/tl.h
file path=usr/include/sys/tnf.h
file path=usr/include/sys/tnf_com.h
file path=usr/include/sys/tnf_probe.h
file path=usr/include/sys/tnf_writer.h
file path=usr/include/sys/todio.h
file path=usr/include/sys/tpicommon.h
file path=usr/include/sys/trap.h
$(i386_ONLY)file path=usr/include/sys/traptrace.h
file path=usr/include/sys/ts.h
file path=usr/include/sys/tsol/label.h
file path=usr/include/sys/tsol/label_macro.h
file path=usr/include/sys/tsol/priv.h
file path=usr/include/sys/tsol/tndb.h
file path=usr/include/sys/tsol/tsyscall.h
file path=usr/include/sys/tspriocntl.h
$(i386_ONLY)file path=usr/include/sys/tss.h
file path=usr/include/sys/ttcompat.h
file path=usr/include/sys/ttold.h
file path=usr/include/sys/tty.h
file path=usr/include/sys/ttychars.h
file path=usr/include/sys/ttydev.h
$(sparc_ONLY)file path=usr/include/sys/ttymux.h
$(sparc_ONLY)file path=usr/include/sys/ttymuxuser.h
file path=usr/include/sys/tuneable.h
file path=usr/include/sys/turnstile.h
file path=usr/include/sys/types.h
file path=usr/include/sys/types32.h
file path=usr/include/sys/tzfile.h
file path=usr/include/sys/u8_textprep.h
file path=usr/include/sys/uadmin.h
$(i386_ONLY)file path=usr/include/sys/ucode.h
file path=usr/include/sys/ucontext.h
file path=usr/include/sys/uio.h
file path=usr/include/sys/ulimit.h
file path=usr/include/sys/un.h
file path=usr/include/sys/unistd.h
file path=usr/include/sys/user.h
file path=usr/include/sys/ustat.h
file path=usr/include/sys/utime.h
file path=usr/include/sys/utrap.h
file path=usr/include/sys/utsname.h
file path=usr/include/sys/utssys.h
file path=usr/include/sys/uuid.h
file path=usr/include/sys/va_impl.h
file path=usr/include/sys/va_list.h
```

```
1580 file path=usr/include/sys/var.h
1581 file path=usr/include/sys/varargs.h
1582 file path=usr/include/sys/vfs.h
1583 file path=usr/include/sys/vfs_opreg.h
1584 file path=usr/include/sys/vfstab.h
1585 file path=usr/include/sys/videodev2.h
1586 file path=usr/include/sys/visual_io.h
1587 file path=usr/include/sys/vm.h
1588 file path=usr/include/sys/vm_usage.h
1589 file path=usr/include/sys/vmem.h
1590 file path=usr/include/sys/vmem_impl.h
1591 file path=usr/include/sys/vmem_impl_user.h
1592 file path=usr/include/sys/vmparam.h
1593 file path=usr/include/sys/vmsystm.h
1594 file path=usr/include/sys/vnode.h
1595 file path=usr/include/sys/vt.h
1596 file path=usr/include/sys/vtdaemon.h
1597 file path=usr/include/sys/vtoc.h
1598 file path=usr/include/sys/vtrace.h
1599 file path=usr/include/sys/vuid_event.h
1600 file path=usr/include/sys/vuid_queue.h
1601 file path=usr/include/sys/vuid_state.h
1602 file path=usr/include/sys/vuid_store.h
1603 file path=usr/include/sys/vuid_wheel.h
1604 file path=usr/include/sys/wait.h
1605 file path=usr/include/sys/waitq.h
1606 file path=usr/include/sys/watchpoint.h
1607 $(i386_ONLY)file path=usr/include/sys/x86_archext.h
1608 $(i386_ONLY)file path=usr/include/sys/xen_errno.h
1609 file path=usr/include/sys/xti_inet.h
1610 file path=usr/include/sys/xti_osi.h
1611 file path=usr/include/sys/xti_xtiopt.h
1612 file path=usr/include/sys/zcons.h
1613 file path=usr/include/sys/zmod.h
1614 file path=usr/include/sys/zone.h
1615 $(sparc_ONLY)file path=usr/include/sys/zsdev.h
1616 file path=usr/include/sysexits.h
1617 file path=usr/include/syslog.h
1618 file path=usr/include/tar.h
1619 file path=usr/include/tcpd.h
1620 file path=usr/include/term.h
1621 file path=usr/include/termcap.h
1622 file path=usr/include/termio.h
1623 file path=usr/include/termios.h
1624 file path=usr/include/thread.h
1625 file path=usr/include/thread_db.h
1626 file path=usr/include/time.h
1627 file path=usr/include/tiuser.h
1628 file path=usr/include/tsol/label.h
1629 file path=usr/include/tzfile.h
1630 file path=usr/include/ucontext.h
1631 file path=usr/include/ucred.h
1632 file path=usr/include/uid_stp.h
1633 file path=usr/include/ulimit.h
1634 file path=usr/include/umem.h
1635 file path=usr/include/umem_impl.h
1636 file path=usr/include/unctrl.h
1637 file path=usr/include/unistd.h
1638 file path=usr/include/user_attr.h
1639 file path=usr/include/userdefs.h
1640 file path=usr/include/ustat.h
1641 file path=usr/include/utility.h
1642 file path=usr/include/utime.h
1643 file path=usr/include/utmp.h
1644 file path=usr/include/utmpx.h
1645 file path=usr/include/uuid/uuid.h
```



```
1646 $(sparc_ONLY)file path=usr/include/v7/sys/machpcb.h
1647 $(sparc_ONLY)file path=usr/include/v7/sys/machtrap.h
1648 $(sparc_ONLY)file path=usr/include/v7/sys/mutex_impl.h
1649 $(sparc_ONLY)file path=usr/include/v7/sys/privregs.h
1650 $(sparc_ONLY)file path=usr/include/v7/sys/prom_isa.h
1651 $(sparc_ONLY)file path=usr/include/v7/sys/psr.h
1652 $(sparc_ONLY)file path=usr/include/v7/sys/traptrace.h
1653 $(sparc_ONLY)file path=usr/include/v9/sys/asi.h
1654 $(sparc_ONLY)file path=usr/include/v9/sys/machpcb.h
1655 $(sparc_ONLY)file path=usr/include/v9/sys/machtrap.h
1656 $(sparc_ONLY)file path=usr/include/v9/sys/membar.h
1657 $(sparc_ONLY)file path=usr/include/v9/sys/mutex_impl.h
1658 $(sparc_ONLY)file path=usr/include/v9/sys/privregs.h
1659 $(sparc_ONLY)file path=usr/include/v9/sys/prom_isa.h
1660 $(sparc_ONLY)file path=usr/include/v9/sys/psr_compat.h
1661 $(sparc_ONLY)file path=usr/include/v9/sys/vis_simulator.h
1662 file path=usr/include/valtools.h
1663 file path=usr/include/values.h
1664 file path=usr/include/varargs.h
1665 file path=usr/include/vm/anon.h
1666 file path=usr/include/vm/as.h
1667 file path=usr/include/vm/faultcode.h
1668 file path=usr/include/vm/hat.h
1669 file path=usr/include/vm/kpm.h
1670 file path=usr/include/vm/page.h
1671 file path=usr/include/vm/pvn.h
1672 file path=usr/include/vm/rm.h
1673 file path=usr/include/vm/seg.h
1674 file path=usr/include/vm/seg_dev.h
1675 file path=usr/include/vm/seg_enum.h
1676 file path=usr/include/vm/seg_kmem.h
1677 file path=usr/include/vm/seg_kp.h
1678 file path=usr/include/vm/seg_kpm.h
1679 file path=usr/include/vm/seg_map.h
1680 file path=usr/include/vm/seg_spt.h
1681 file path=usr/include/vm/seg_vn.h
1682 file path=usr/include/vm/vpage.h
1683 file path=usr/include/vm/vpm.h
1684 file path=usr/include/volmgt.h
1685 file path=usr/include/wait.h
1686 file path=usr/include/wchar.h
1687 file path=usr/include/wchar_impl.h
1688 file path=usr/include/wctype.h
1689 file path=usr/include/widec.h
1690 file path=usr/include/wordexp.h
1691 file path=usr/include/xti.h
1692 file path=usr/include/xti_inet.h
1693 file path=usr/include/zone.h
1694 file path=usr/include/zonestat.h
1695 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/acpidev.h
1696 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/amd_iommu.h
1697 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/asm_misc.h
1698 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/clock.h
1699 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/cram.h
1700 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/ddi_subrdefs.h
1701 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/debug_info.h
1702 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/fastboot.h
1703 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/mach_mmu.h
1704 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machclock.h
1705 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machcpuvar.h
1706 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machparam.h
1707 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machprivregs.h
1708 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machsystm.h
1709 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machthread.h
1710 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/memnode.h
1711 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/pc_mmu.h
```

```
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_defs.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_modctl.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_types.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/rm_platter.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/sbd_ioctl.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/smp_impldefs.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/vm_machparam.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/x_call.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/xc_levels.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/xsvc.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_i86.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_pte.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/hment.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/htable.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/kboot_mmu.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/balloon.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/machprivregs.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xen_mmu.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xpv_impl.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/vm/seg_mf.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ac.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/async.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cheetahregs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cherrystone.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/clock.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cmp.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpc_ultra.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpr_impl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_impl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_sgnblk_defs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cvc.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/daktari.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ddi_subrdefs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/dvma.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ecc_kstat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/eeprom.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_gen.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue250.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue450.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/environ.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/errclassify.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/fhc.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/gpio_87317.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/hpc3130_events.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/hpc3130.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/i2c_client.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/lm75.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/max1617.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/pcf8591.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/ssc050.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/misc/i2c_svc.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/idprom.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intr.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intreg.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iocache.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iommu.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ivintr.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/lom_io.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machasi.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machclock.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machcpuvar.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machparam.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machsystm.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machthread.h
```



```
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mem_cache.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memlist_plat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memnode.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mmu.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/nexusdebug.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_hwdesc.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_module.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_debug.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_plat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/pte.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sbd_ioctl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scb.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scsb_led.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/simmstat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/spitregs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sram.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/starfire.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sun4asi.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysctrl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysioerr.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysiosbus.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/tod.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/todmostek.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/trapstat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/traptrace.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vis.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vm_machparam.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/x_call.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/xc_impl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/zsmach.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/vm/hat_sfmmu.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/vm/mach_sfmmu.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/clock.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cmp.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpc_ultra.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpu_sgnblk_defs.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ddi_subrdefs.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_pri.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_snmp.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/dvma.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/eeprom.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/fcode.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hsvc.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hypervisor_api.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/idprom.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intr.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intreg.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ivintr.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machasi.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machclock.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machcpuvar.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machintreg.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machparam.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machsystm.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machthread.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memlist_plat.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memnode.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/mmu.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/nexusdebug.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagaraasi.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagararegs.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ntwdt.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pri.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_debug.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_plat.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pte.h
```

```
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/qcn.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/scb.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/soft_state.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/sun4asi.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/tod.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/trapstat.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/traptrace.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vis.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vm_machparam.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/x_call.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/xc_impl.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/zsmach.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/vm/hat_sfmmu.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/vm/mach_sfmmu.h
file path=usr/share/man/man3head/acct.3head
file path=usr/share/man/man3head/acct.h.3head
file path=usr/share/man/man3head/aio.3head
file path=usr/share/man/man3head/aio.h.3head
file path=usr/share/man/man3head/ar.3head
file path=usr/share/man/man3head/ar.h.3head
file path=usr/share/man/man3head/archives.3head
file path=usr/share/man/man3head/archives.h.3head
file path=usr/share/man/man3head/assert.3head
file path=usr/share/man/man3head/assert.h.3head
file path=usr/share/man/man3head/complex.3head
file path=usr/share/man/man3head/complex.h.3head
file path=usr/share/man/man3head/cpio.3head
file path=usr/share/man/man3head/cpio.h.3head
file path=usr/share/man/man3head/dirent.3head
file path=usr/share/man/man3head/dirent.h.3head
file path=usr/share/man/man3head/errno.3head
file path=usr/share/man/man3head/errno.h.3head
file path=usr/share/man/man3head/fcntl.3head
file path=usr/share/man/man3head/fcntl.h.3head
file path=usr/share/man/man3head/fenv.3head
file path=usr/share/man/man3head/fenv.h.3head
file path=usr/share/man/man3head/float.3head
file path=usr/share/man/man3head/float.h.3head
file path=usr/share/man/man3head/floatingpoint.3head
file path=usr/share/man/man3head/floatingpoint.h.3head
file path=usr/share/man/man3head/fmtmsg.3head
file path=usr/share/man/man3head/fmtmsg.h.3head
file path=usr/share/man/man3head/fnmatch.3head
file path=usr/share/man/man3head/fnmatch.h.3head
file path=usr/share/man/man3head/ftw.3head
file path=usr/share/man/man3head/ftw.h.3head
file path=usr/share/man/man3head/glob.3head
file path=usr/share/man/man3head/glob.h.3head
file path=usr/share/man/man3head/grp.3head
file path=usr/share/man/man3head/grp.h.3head
file path=usr/share/man/man3head/iconv.3head
file path=usr/share/man/man3head/iconv.h.3head
file path=usr/share/man/man3head/if.3head
file path=usr/share/man/man3head/if.h.3head
file path=usr/share/man/man3head/in.3head
file path=usr/share/man/man3head/in.h.3head
file path=usr/share/man/man3head/inet.3head
file path=usr/share/man/man3head/inet.h.3head
file path=usr/share/man/man3head/inttypes.3head
file path=usr/share/man/man3head/inttypes.h.3head
file path=usr/share/man/man3head/ipc.3head
file path=usr/share/man/man3head/ipc.h.3head
file path=usr/share/man/man3head/iso646.3head
file path=usr/share/man/man3head/iso646.h.3head
file path=usr/share/man/man3head/langinfo.3head
file path=usr/share/man/man3head/langinfo.h.3head
```



```
file path=usr/share/man/man3head/libgen.3head
file path=usr/share/man/man3head/libgen.h.3head
file path=usr/share/man/man3head/libintl.3head
file path=usr/share/man/man3head/libintl.h.3head
file path=usr/share/man/man3head/limits.3head
file path=usr/share/man/man3head/limits.h.3head
file path=usr/share/man/man3head/locale.3head
file path=usr/share/man/man3head/locale.h.3head
file path=usr/share/man/man3head/math.3head
file path=usr/share/man/man3head/math.h.3head
file path=usr/share/man/man3head/mman.3head
file path=usr/share/man/man3head/mman.h.3head
file path=usr/share/man/man3head/monetary.3head
file path=usr/share/man/man3head/monetary.h.3head
file path=usr/share/man/man3head/mqueue.3head
file path=usr/share/man/man3head/mqueue.h.3head
file path=usr/share/man/man3head/msg.3head
file path=usr/share/man/man3head/msg.h.3head
file path=usr/share/man/man3head/ndbm.3head
file path=usr/share/man/man3head/ndbm.h.3head
file path=usr/share/man/man3head/netdb.3head
file path=usr/share/man/man3head/netdb.h.3head
file path=usr/share/man/man3head/nl_types.3head
file path=usr/share/man/man3head/nl_types.h.3head
file path=usr/share/man/man3head/poll.3head
file path=usr/share/man/man3head/poll.h.3head
file path=usr/share/man/man3head/pthread.3head
file path=usr/share/man/man3head/pthread.h.3head
file path=usr/share/man/man3head/pwd.3head
file path=usr/share/man/man3head/pwd.h.3head
file path=usr/share/man/man3head/regex.3head
file path=usr/share/man/man3head/regex.h.3head
file path=usr/share/man/man3head/resource.3head
file path=usr/share/man/man3head/resource.h.3head
file path=usr/share/man/man3head/sched.3head
file path=usr/share/man/man3head/sched.h.3head
file path=usr/share/man/man3head/search.3head
file path=usr/share/man/man3head/search.h.3head
file path=usr/share/man/man3head/select.3head
file path=usr/share/man/man3head/select.h.3head
file path=usr/share/man/man3head/sem.3head
file path=usr/share/man/man3head/sem.h.3head
file path=usr/share/man/man3head/semaphore.3head
file path=usr/share/man/man3head/semaphore.h.3head
file path=usr/share/man/man3head/setjmp.3head
file path=usr/share/man/man3head/setjmp.h.3head
file path=usr/share/man/man3head/shm.3head
file path=usr/share/man/man3head/shm.h.3head
file path=usr/share/man/man3head/siginfo.3head
file path=usr/share/man/man3head/siginfo.h.3head
file path=usr/share/man/man3head/signal.3head
file path=usr/share/man/man3head/signal.h.3head
file path=usr/share/man/man3head/socket.3head
file path=usr/share/man/man3head/socket.h.3head
file path=usr/share/man/man3head/spawn.3head
file path=usr/share/man/man3head/spawn.h.3head
file path=usr/share/man/man3head/stat.3head
file path=usr/share/man/man3head/stat.h.3head
file path=usr/share/man/man3head/statvfs.3head
file path=usr/share/man/man3head/statvfs.h.3head
file path=usr/share/man/man3head/stdbool.3head
file path=usr/share/man/man3head/stdbool.h.3head
file path=usr/share/man/man3head/stddef.3head
file path=usr/share/man/man3head/stddef.h.3head
file path=usr/share/man/man3head/stdint.3head
file path=usr/share/man/man3head/stdint.h.3head
```

```
1976 file path=usr/share/man/man3head/stdio.3head
1977 file path=usr/share/man/man3head/stdio.h.3head
1978 file path=usr/share/man/man3head/stdlib.3head
1979 file path=usr/share/man/man3head/stdlib.h.3head
1980 file path=usr/share/man/man3head/string.3head
1981 file path=usr/share/man/man3head/string.h.3head
1982 file path=usr/share/man/man3head/strings.3head
1983 file path=usr/share/man/man3head/strings.h.3head
1984 file path=usr/share/man/man3head/stropts.3head
1985 file path=usr/share/man/man3head/stropts.h.3head
1986 file path=usr/share/man/man3head/syslog.3head
1987 file path=usr/share/man/man3head/syslog.h.3head
1988 file path=usr/share/man/man3head/tar.3head
1989 file path=usr/share/man/man3head/tar.h.3head
1990 file path=usr/share/man/man3head/tcp.3head
1991 file path=usr/share/man/man3head/tcp.h.3head
1992 file path=usr/share/man/man3head/termios.3head
1993 file path=usr/share/man/man3head/termios.h.3head
1994 file path=usr/share/man/man3head/tgmath.3head
1995 file path=usr/share/man/man3head/tgmath.h.3head
1996 file path=usr/share/man/man3head/time.3head
1997 file path=usr/share/man/man3head/time.h.3head
1998 file path=usr/share/man/man3head/timeb.3head
1999 file path=usr/share/man/man3head/timeb.h.3head
2000 file path=usr/share/man/man3head/times.3head
2001 file path=usr/share/man/man3head/times.h.3head
2002 file path=usr/share/man/man3head/types.3head
2003 file path=usr/share/man/man3head/types.h.3head
2004 file path=usr/share/man/man3head/types32.3head
2005 file path=usr/share/man/man3head/types32.h.3head
2006 file path=usr/share/man/man3head/ucontext.3head
2007 file path=usr/share/man/man3head/ucontext.h.3head
2008 file path=usr/share/man/man3head/uio.3head
2009 file path=usr/share/man/man3head/uio.h.3head
2010 file path=usr/share/man/man3head/ulimit.3head
2011 file path=usr/share/man/man3head/ulimit.h.3head
2012 file path=usr/share/man/man3head/un.3head
2013 file path=usr/share/man/man3head/un.h.3head
2014 file path=usr/share/man/man3head/unistd.3head
2015 file path=usr/share/man/man3head/unistd.h.3head
2016 file path=usr/share/man/man3head/utime.3head
2017 file path=usr/share/man/man3head/utime.h.3head
2018 file path=usr/share/man/man3head/utmpx.3head
2019 file path=usr/share/man/man3head/utmpx.h.3head
2020 file path=usr/share/man/man3head/utsname.3head
2021 file path=usr/share/man/man3head/utsname.h.3head
2022 file path=usr/share/man/man3head/values.3head
2023 file path=usr/share/man/man3head/values.h.3head
2024 file path=usr/share/man/man3head/wait.3head
2025 file path=usr/share/man/man3head/wait.h.3head
2026 file path=usr/share/man/man3head/wchar.3head
2027 file path=usr/share/man/man3head/wchar.h.3head
2028 file path=usr/share/man/man3head/wctype.3head
2029 file path=usr/share/man/man3head/wctype.h.3head
2030 file path=usr/share/man/man3head/wordexp.3head
2031 file path=usr/share/man/man3head/wordexp.h.3head
2032 file path=usr/share/man/man4/note.4
2033 file path=usr/share/man/man5/prof.5
2034 file path=usr/share/man/man7i/cdio.7i
2035 file path=usr/share/man/man7i/dkio.7i
2036 file path=usr/share/man/man7i/fbio.7i
2037 file path=usr/share/man/man7i/fdio.7i
2038 file path=usr/share/man/man7i/hdio.7i
2039 file path=usr/share/man/man7i/iec61883.7i
2040 file path=usr/share/man/man7i/mhd.7i
2041 file path=usr/share/man/man7i/mtio.7i
```



```
2042 file path=usr/share/man/man7i/prnio.7i
2043 file path=usr/share/man/man7i/quotactl.7i
2044 file path=usr/share/man/man7i/sesio.7i
2045 file path=usr/share/man/man7i/sockio.7i
2046 file path=usr/share/man/man7i/streamio.7i
2047 file path=usr/share/man/man7i/termio.7i
2048 file path=usr/share/man/man7i/termiox.7i
2049 file path=usr/share/man/man7i/uscsi.7i
2050 file path=usr/share/man/man7i/visual_io.7i
2051 file path=usr/share/man/man7i/vt.7i
2052 file path=usr/xpg4/include/curses.h
2053 file path=usr/xpg4/include/term.h
2054 file path=usr/xpg4/include/unctrl.h
2055 legacy pkg=SUNWhea \
2056     desc="SunOS C/C++ header files for general development of software" \
2057     name="SunOS Header Files"
2058 license cr_Sun license=cr_Sun
2059 license lic_CDDL license=lic_CDDL
2060 license license_in_headers license=license_in_headers
2061 license usr/src/lib/pkcs11/include/THIRDPARTYLICENSE \
2062     license=usr/src/lib/pkcs11/include/THIRDPARTYLICENSE
2063 link path=usr/include/iso/assert_iso.h target=../assert.h
2064 link path=usr/include/iso/errno_iso.h target=../errno.h
2065 link path=usr/include/iso/float_iso.h target=../float.h
2066 link path=usr/include/iso/iso646_iso.h target=../iso646.h
2067 $(sparc_ONLY)link path=usr/platform/SUNW,A70/include target=../sun4u/include
2068 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T12/include \
2069     target=../sun4u/include
2070 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T4/include \
2071     target=../sun4u/include
2072 $(sparc_ONLY)link path=usr/platform/SUNW,SPARC-Enterprise/include \
2073     target=../sun4u/include
2074 $(sparc_ONLY)link path=usr/platform/SUNW,Serverblade1/include \
2075     target=../sun4u/include
2076 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-100/include \
2077     target=../sun4u/include
2078 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1000/include \
2079     target=../sun4u/include
2080 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1500/include \
2081     target=../sun4u/include
2082 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-2500/include \
2083     target=../sun4u/include
2084 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-15000/include \
2085     target=../sun4u/include
2086 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-280R/include \
2087     target=../sun4u/include
2088 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-480R/include \
2089     target=../sun4u/include
2090 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-880/include \
2091     target=../sun4u/include
2092 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V215/include \
2093     target=../sun4u/include
2094 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V240/include \
2095     target=../sun4u/include
2096 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V250/include \
2097     target=../sun4u/include
2098 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V440/include \
2099     target=../sun4u/include
2100 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V445/include \
2101     target=../sun4u/include
2102 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V490/include \
2103     target=../sun4u/include
2104 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V890/include \
2105     target=../sun4u/include
2106 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire/include \
2107     target=../sun4u/include
```

```
2108 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-2/include \
2109     target=../sun4u/include
2110 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-250/include \
2111     target=../sun4u/include
2112 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-4/include \
2113     target=../sun4u/include
2114 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise-10000/include \
2115     target=../sun4u/include
2116 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise/include \
2117     target=../sun4u/include
2118 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40/include \
2119     target=../sun4u/include
2120 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60/include \
2121     target=../sun4u/include
2122 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIi-Netract/include \
2123     target=../sun4u/include
2124 $(i386_ONLY)link path=usr/share/src/uts/i86pc/sys \
2125     target=../../../../platform/i86pc/include/sys
2126 $(i386_ONLY)link path=usr/share/src/uts/i86pc/vm \
2127     target=../../../../platform/i86pc/include/vm
2128 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/sys \
2129     target=../../../../platform/i86xpv/include/sys
2130 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/vm \
2131     target=../../../../platform/i86xpv/include/vm
2132 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/sys \
2133     target=../../../../platform/sun4u/include/sys
2134 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/vm \
2135     target=../../../../platform/sun4u/include/vm
2136 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/sys \
2137     target=../../../../platform/sun4v/include/sys
2138 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/vm \
2139     target=../../../../platform/sun4v/include/vm
```

```
**********************************************************
   45632 Mon Jul  9 14:38:11 2012
new/usr/src/pkg/manifests/system-kernel.mf
dccp: manifest
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
#

#
# The default for payload-bearing actions in this package is to appear in the
# global zone only.  See the include file for greater detail, as well as
# information about overriding the defaults.
#
<include global_zone_only_component>
<include system-kernel.man1m.inc>
<include system-kernel.man2.inc>
<include system-kernel.man4.inc>
<include system-kernel.man5.inc>
<include system-kernel.man7.inc>
<include system-kernel.man7d.inc>
<include system-kernel.man7fs.inc>
<include system-kernel.man7m.inc>
<include system-kernel.man7p.inc>
<include system-kernel.man9.inc>
<include system-kernel.man9e.inc>
<include system-kernel.man9f.inc>
<include system-kernel.man9p.inc>
<include system-kernel.man9s.inc>
set name=pkg.fmri value=pkg:/system/kernel@$(PKGVERS)
set name=pkg.description \
    value="core kernel software for a specific instruction-set architecture"
set name=pkg.summary value="Core Solaris Kernel"
set name=info.classification value=org.opensolaris.category.2008:System/Core
set name=variant.arch value=$(ARCH)
dir path=boot group=sys
$(i386_ONLY)dir path=boot/acpi group=sys
$(i386_ONLY)dir path=boot/acpi/tables group=sys
dir path=boot/solaris group=sys
dir path=boot/solaris/bin group=sys
dir path=etc group=sys
dir path=etc/crypto group=sys
dir path=etc/sock2path.d group=sys
dir path=kernel group=sys
$(i386_ONLY)dir path=kernel/$(ARCH64) group=sys
```



```
dir path=kernel/crypto group=sys
dir path=kernel/crypto/$(ARCH64) group=sys
dir path=kernel/dacf group=sys
dir path=kernel/dacf/$(ARCH64) group=sys
dir path=kernel/drv group=sys
dir path=kernel/drv/$(ARCH64) group=sys
dir path=kernel/exec group=sys
dir path=kernel/exec/$(ARCH64) group=sys
dir path=kernel/fs group=sys
dir path=kernel/fs/$(ARCH64) group=sys
dir path=kernel/ipp group=sys
dir path=kernel/ipp/$(ARCH64) group=sys
dir path=kernel/kiconv group=sys
dir path=kernel/kiconv/$(ARCH64) group=sys
dir path=kernel/mac group=sys
dir path=kernel/mac/$(ARCH64) group=sys
dir path=kernel/misc group=sys
dir path=kernel/misc/$(ARCH64) group=sys
dir path=kernel/misc/scsi_vhci group=sys
dir path=kernel/misc/scsi_vhci/$(ARCH64) group=sys
dir path=kernel/sched group=sys
dir path=kernel/sched/$(ARCH64) group=sys
dir path=kernel/socketmod group=sys
dir path=kernel/socketmod/$(ARCH64) group=sys
dir path=kernel/strmod group=sys
dir path=kernel/strmod/$(ARCH64) group=sys
dir path=kernel/sys group=sys
dir path=kernel/sys/$(ARCH64) group=sys
dir path=lib
dir path=lib/svc
dir path=lib/svc/manifest group=sys
dir path=lib/svc/manifest/system group=sys
dir path=lib/svc/method
dir path=usr/share/man
dir path=usr/share/man/man1m
dir path=usr/share/man/man2
dir path=usr/share/man/man3
dir path=usr/share/man/man4
dir path=usr/share/man/man5
dir path=usr/share/man/man7d
dir path=usr/share/man/man7fs
dir path=usr/share/man/man7m
dir path=usr/share/man/man7p
dir path=usr/share/man/man9
dir path=usr/share/man/man9e
dir path=usr/share/man/man9f
dir path=usr/share/man/man9p
dir path=usr/share/man/man9s
$(i386_ONLY)driver name=acpi_drv perms="* 0666 root sys"
driver name=aggr perms="* 0666 root sys"
driver name=arp perms="arp 0666 root sys"
driver name=bl perms="* 0666 root sys"
driver name=bridge clone_perms="bridge 0666 root sys" \
    policy="read_priv_set=net_rawaccess write_priv_set=net_rawaccess"
$(sparc_ONLY)driver name=bscbus alias=SUNW,bscbus
$(i386_ONLY)driver name=bscbus alias=SVI0101
$(sparc_ONLY)driver name=bscv alias=SUNW,bscv perms="* 0644 root sys"
$(i386_ONLY)driver name=bscv
driver name=clone
driver name=cn perms="* 0620 root tty"
driver name=conskbd perms="kbd 0666 root sys"
driver name=consms perms="mouse 0666 root sys"
driver name=cpuid perms="self 0644 root sys"
$(i386_ONLY)driver name=cpunex alias=cpus
driver name=crypto perms="crypto 0666 root sys"
driver name=cryptoadm perms="cryptoadm 0644 root sys"
```

```
$(sparc_ONLY)driver name=dad alias=ide-disk perms="* 0640 root sys"
driver name=dccp perms="dccp 0666 root sys"
driver name=dccp6 perms="dccp6 0666 root sys"
#endif /* ! codereview */
driver name=devinfo perms="devinfo 0640 root sys" \
    perms="devinfo,ro 0444 root sys"
driver name=dld perms="* 0666 root sys"
driver name=dlpistub perms="* 0666 root sys"
$(sparc_ONLY)driver name=i8042 alias=8042
$(i386_ONLY)driver name=i8042
driver name=icmp perms="icmp 0666 root sys" \
    policy="read_priv_set=net_icmpaccess write_priv_set=net_icmpaccess"
driver name=icmp6 perms="icmp6 0666 root sys" \
    policy="read_priv_set=net_icmpaccess write_priv_set=net_icmpaccess"
$(i386_ONLY)driver name=intel_nb5000 \
    alias=pci8086,25c0 \
    alias=pci8086,25d0 \
    alias=pci8086,25d4 \
    alias=pci8086,25d8 \
    alias=pci8086,3600 \
    alias=pci8086,4000 \
    alias=pci8086,4001 \
    alias=pci8086,4003 \
    alias=pci8086,65c0
$(i386_ONLY)driver name=intel_nhm \
    alias=pci8086,3423 \
    alias=pci8086,372a
$(i386_ONLY)driver name=intel_nhmex alias=pci8086,3438
driver name=ip perms="ip 0666 root sys" \
    policy="read_priv_set=net_rawaccess write_priv_set=net_rawaccess"
driver name=ip6 perms="ip6 0666 root sys" \
    policy="read_priv_set=net_rawaccess write_priv_set=net_rawaccess"
driver name=ipnet perms="lo0 0666 root sys" \
    policy="read_priv_set=net_observability write_priv_set=net_observability"
driver name=ippctl
driver name=ipsecah perms="ipsecah 0666 root sys" \
    policy="read_priv_set=sys_ip_config write_priv_set=sys_ip_config"
driver name=ipsecesp perms="ipsecesp 0666 root sys" \
    policy="read_priv_set=sys_ip_config write_priv_set=sys_ip_config"
driver name=iptun
driver name=iwscn
driver name=kb8042 alias=pnpPNP,303
driver name=keysock perms="keysock 0666 root sys" \
    policy="read_priv_set=sys_ip_config write_priv_set=sys_ip_config"
driver name=kmdb
driver name=kssl perms="* 0666 root sys"
driver name=llc1 clone_perms="llc1 0666 root sys"
driver name=lofi perms="* 0600 root sys" perms="ctl 0644 root sys"
driver name=log perms="conslog 0666 root sys" perms="log 0640 root sys"
$(i386_ONLY)driver name=mc-amd \
    alias=pci1022,1100 \
    alias=pci1022,1101 \
    alias=pci1022,1102
driver name=mm perms="allkmem 0600 root sys" perms="kmem 0640 root sys" \
    perms="mem 0640 root sys" perms="null 0666 root sys" \
    perms="zero 0666 root sys" \
    policy="allkmem read_priv_set=all write_priv_set=all" \
    policy="kmem read_priv_set=none write_priv_set=all" \
    policy="mem read_priv_set=none write_priv_set=all"
driver name=mouse8042 alias=pnpPNP,f03
$(i386_ONLY)driver name=mpt class=scsi \
    alias=pci1000,30 \
    alias=pci1000,50 \
    alias=pci1000,54 \
    alias=pci1000,56 \
    alias=pci1000,58 \
```



```
    alias=pci1000,62 \
    alias=pciex1000,56 \
    alias=pciex1000,58 \
    alias=pciex1000,62
driver name=nulldriver \
    alias=scsa,nodev \
    alias=scsa,probe
driver name=openeepr perms="openprom 0640 root sys" policy=write_priv_set=all
driver name=options
$(sparc_ONLY)driver name=pci_pci class=pci \
    alias=pci1011,1 \
    alias=pci1011,21 \
    alias=pci1011,24 \
    alias=pci1011,25 \
    alias=pci1011,26 \
    alias=pci1014,22 \
    alias=pciclass,060400
$(i386_ONLY)driver name=pci_pci class=pci \
    alias=pci1011,1 \
    alias=pci1011,21 \
    alias=pci1014,22 \
    alias=pciclass,060400 \
    alias=pciclass,060401
$(sparc_ONLY)driver name=pcieb \
    alias=pciex108e,9010 \
    alias=pciex108e,9020 \
    alias=pciex10b5,8114 \
    alias=pciex10b5,8516 \
    alias=pciex10b5,8517 \
    alias=pciex10b5,8518 \
    alias=pciex10b5,8532 \
    alias=pciex10b5,8533 \
    alias=pciex10b5,8548 \
    alias=pciexclass,060400
$(i386_ONLY)driver name=pcieb \
    alias=pciexclass,060400 \
    alias=pciexclass,060401
$(sparc_ONLY)driver name=pcieb_bcm alias=pciex1166,103
driver name=physmem perms="* 0600 root sys"
driver name=poll perms="* 0666 root sys"
$(sparc_ONLY)driver name=power alias=ali1535d+-power
$(i386_ONLY)driver name=power
driver name=pseudo alias=zconsnex
driver name=ptc perms="* 0666 root sys"
driver name=ptsl perms="* 0666 root sys"
$(sparc_ONLY)driver name=ramdisk alias=SUNW,ramdisk perms="* 0600 root sys" \
    perms="ctl 0644 root sys"
$(i386_ONLY)driver name=ramdisk perms="* 0600 root sys" \
    perms="ctl 0644 root sys"
driver name=random perms="* 0644 root sys" policy=write_priv_set=sys_devices
driver name=rts perms="rts 0666 root sys"
driver name=sad perms="admin 0666 root sys" perms="user 0666 root sys"
driver name=scsi_vhci class=scsi-self-identifying perms="* 0666 root sys" \
    policy="devctl write_priv_set=sys_devices"
$(sparc_ONLY)driver name=sd perms="* 0640 root sys" \
    alias=ide-cdrom \
    alias=scsiclass,00 \
    alias=scsiclass,05
$(i386_ONLY)driver name=sd perms="* 0640 root sys" \
    alias=scsiclass,00 \
    alias=scsiclass,05
driver name=sgen perms="* 0600 root sys" \
    alias=scsa,08.bfcp \
    alias=scsa,08.bvhci
driver name=simnet clone_perms="simnet 0666 root sys" perms="* 0666 root sys"
$(i386_ONLY)driver name=smbios perms="smbios 0444 root sys"
```

```
260 driver name=softmac
261 driver name=spdsock perms="spdsock 0666 root sys" \
262     policy="read_priv_set=sys_ip_config write_priv_set=sys_ip_config"
263 driver name=st alias=scsiclass,01 perms="* 0666 root sys"
264 driver name=sy perms="tty 0666 root tty"
265 driver name=sysevent perms="* 0600 root sys"
266 driver name=sysmsg perms="msglog 0600 root sys" perms="sysmsg 0600 root sys"
267 driver name=tcp perms="tcp 0666 root sys"
268 driver name=tcp6 perms="tcp6 0666 root sys"
269 driver name=tl perms="* 0666 root sys" clone_perms="ticlts 0666 root sys" \
270     clone_perms="ticots 0666 root sys" clone_perms="ticotsord 0666 root sys"
271 $(sparc_ONLY)driver name=ttymux alias=multiplexer
272 $(i386_ONLY)driver name=tzmon
273 $(sparc_ONLY)driver name=uata \
274     alias=pci1095,646 \
275     alias=pci1095,649 \
276     alias=pci1095,680 \
277     alias=pci10b9,5229 \
278     alias=pci10b9,5288 class=dada class=scsi
279 $(i386_ONLY)driver name=ucode perms="* 0644 root sys"
280 driver name=udp perms="udp 0666 root sys"
281 driver name=udp6 perms="udp6 0666 root sys"
282 $(i386_ONLY)driver name=vgatext \
283     alias=pciclass,000100 \
284     alias=pciclass,030000 \
285     alias=pciclass,030001 \
286     alias=pnpPNP,900
287 driver name=vnic clone_perms="vnic 0666 root sys" perms="* 0666 root sys"
288 driver name=wc perms="* 0600 root sys"
289 $(i386_ONLY)file path=boot/solaris/bin/create_diskmap group=sys mode=0555
290 file path=boot/solaris/bin/create_ramdisk group=sys mode=0555
291 file path=boot/solaris/bin/extract_boot_filelist group=sys mode=0555
292 $(i386_ONLY)file path=boot/solaris/bin/mbr group=sys mode=0555
293 $(i386_ONLY)file path=boot/solaris/bin/symdef group=sys mode=0555
294 $(i386_ONLY)file path=boot/solaris/bin/update_grub group=sys mode=0555
295 file path=boot/solaris/filelist.ramdisk group=sys
296 file path=boot/solaris/filelist.safe group=sys
297 file path=etc/crypto/kcf.conf group=sys \
298     original_name=SUNWckr:etc/crypto/kcf.conf preserve=true
299 file path=etc/name_to_sysnum group=sys \
300     original_name=SUNWckr:etc/name_to_sysnum preserve=renameold
301 file path=etc/sock2path.d/system%2Fkernel group=sys
302 file path=etc/system group=sys original_name=SUNWckr:etc/system preserve=true
303 $(i386_ONLY)file path=kernel/$(ARCH64)/genunix group=sys mode=0755
304 file path=kernel/crypto/$(ARCH64)/aes group=sys mode=0755
305 file path=kernel/crypto/$(ARCH64)/arcfour group=sys mode=0755
306 file path=kernel/crypto/$(ARCH64)/blowfish group=sys mode=0755
307 file path=kernel/crypto/$(ARCH64)/des group=sys mode=0755
308 file path=kernel/crypto/$(ARCH64)/ecc group=sys mode=0755
309 file path=kernel/crypto/$(ARCH64)/md4 group=sys mode=0755
310 file path=kernel/crypto/$(ARCH64)/md5 group=sys mode=0755
311 file path=kernel/crypto/$(ARCH64)/rsa group=sys mode=0755
312 file path=kernel/crypto/$(ARCH64)/sha1 group=sys mode=0755
313 file path=kernel/crypto/$(ARCH64)/sha2 group=sys mode=0755
314 file path=kernel/crypto/$(ARCH64)/swrand group=sys mode=0755
315 $(i386_ONLY)file path=kernel/crypto/aes group=sys mode=0755
316 $(i386_ONLY)file path=kernel/crypto/arcfour group=sys mode=0755
317 $(i386_ONLY)file path=kernel/crypto/blowfish group=sys mode=0755
318 $(i386_ONLY)file path=kernel/crypto/des group=sys mode=0755
319 $(i386_ONLY)file path=kernel/crypto/ecc group=sys mode=0755
320 $(i386_ONLY)file path=kernel/crypto/md4 group=sys mode=0755
321 $(i386_ONLY)file path=kernel/crypto/md5 group=sys mode=0755
322 $(i386_ONLY)file path=kernel/crypto/rsa group=sys mode=0755
323 $(i386_ONLY)file path=kernel/crypto/sha1 group=sys mode=0755
324 $(i386_ONLY)file path=kernel/crypto/sha2 group=sys mode=0755
325 $(i386_ONLY)file path=kernel/crypto/swrand group=sys mode=0755
```



```
326 $(sparc_ONLY)file path=kernel/dacf/$(ARCH64)/consconfig_dacf group=sys \
327     mode=0755
328 file path=kernel/dacf/$(ARCH64)/net_dacf group=sys mode=0755
329 $(i386_ONLY)file path=kernel/dacf/net_dacf group=sys mode=0755
330 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/acpi_drv group=sys
331 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/acpi_toshiba group=sys
332 file path=kernel/drv/$(ARCH64)/aggr group=sys
333 file path=kernel/drv/$(ARCH64)/arp group=sys
334 file path=kernel/drv/$(ARCH64)/bl group=sys
335 file path=kernel/drv/$(ARCH64)/bridge group=sys
336 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/bscbus group=sys
337 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/bscv group=sys
338 file path=kernel/drv/$(ARCH64)/clone group=sys
339 file path=kernel/drv/$(ARCH64)/cn group=sys
340 file path=kernel/drv/$(ARCH64)/conskbd group=sys
341 file path=kernel/drv/$(ARCH64)/consms group=sys
342 file path=kernel/drv/$(ARCH64)/cpuid group=sys
343 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/cpunex group=sys
344 file path=kernel/drv/$(ARCH64)/crypto group=sys
345 file path=kernel/drv/$(ARCH64)/cryptoadm group=sys
346 $(sparc_ONLY)file path=kernel/drv/$(ARCH64)/dad group=sys
347 file path=kernel/drv/$(ARCH64)/dccp group=sys
348 file path=kernel/drv/$(ARCH64)/dccp6 group=sys
349 #endif /* ! codereview */
350 file path=kernel/drv/$(ARCH64)/devinfo group=sys
351 file path=kernel/drv/$(ARCH64)/dld group=sys
352 file path=kernel/drv/$(ARCH64)/dlpistub group=sys
353 file path=kernel/drv/$(ARCH64)/i8042 group=sys
354 file path=kernel/drv/$(ARCH64)/icmp group=sys
355 file path=kernel/drv/$(ARCH64)/icmp6 group=sys
356 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/intel_nb5000 group=sys
357 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/intel_nhm group=sys
358 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/intel_nhmex group=sys
359 file path=kernel/drv/$(ARCH64)/ip group=sys
360 file path=kernel/drv/$(ARCH64)/ip6 group=sys
361 file path=kernel/drv/$(ARCH64)/ipnet group=sys
362 file path=kernel/drv/$(ARCH64)/ippctl group=sys
363 file path=kernel/drv/$(ARCH64)/ipsecah group=sys
364 file path=kernel/drv/$(ARCH64)/ipsecesp group=sys
365 file path=kernel/drv/$(ARCH64)/iptun group=sys
366 file path=kernel/drv/$(ARCH64)/iwscn group=sys
367 file path=kernel/drv/$(ARCH64)/kb8042 group=sys
368 file path=kernel/drv/$(ARCH64)/keysock group=sys
369 file path=kernel/drv/$(ARCH64)/kmdb group=sys
370 file path=kernel/drv/$(ARCH64)/kssl group=sys
371 file path=kernel/drv/$(ARCH64)/llc1 group=sys
372 file path=kernel/drv/$(ARCH64)/lofi group=sys
373 file path=kernel/drv/$(ARCH64)/log group=sys
374 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/mc-amd group=sys
375 file path=kernel/drv/$(ARCH64)/mm group=sys
376 file path=kernel/drv/$(ARCH64)/mouse8042 group=sys
377 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/mpt group=sys
378 file path=kernel/drv/$(ARCH64)/nulldriver group=sys
379 file path=kernel/drv/$(ARCH64)/openeepr group=sys
380 file path=kernel/drv/$(ARCH64)/options group=sys
381 file path=kernel/drv/$(ARCH64)/pci_pci group=sys
382 file path=kernel/drv/$(ARCH64)/pcieb group=sys
383 $(sparc_ONLY)file path=kernel/drv/$(ARCH64)/pcieb_bcm group=sys
384 file path=kernel/drv/$(ARCH64)/physmem group=sys
385 file path=kernel/drv/$(ARCH64)/poll group=sys
386 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/power group=sys
387 file path=kernel/drv/$(ARCH64)/pseudo group=sys
388 file path=kernel/drv/$(ARCH64)/ptc group=sys
389 file path=kernel/drv/$(ARCH64)/ptsl group=sys
390 file path=kernel/drv/$(ARCH64)/ramdisk group=sys
391 file path=kernel/drv/$(ARCH64)/random group=sys
```

```
392 file path=kernel/drv/$(ARCH64)/rts group=sys
393 file path=kernel/drv/$(ARCH64)/sad group=sys
394 file path=kernel/drv/$(ARCH64)/scsi_vhci group=sys
395 file path=kernel/drv/$(ARCH64)/sd group=sys
396 file path=kernel/drv/$(ARCH64)/sgen group=sys
397 file path=kernel/drv/$(ARCH64)/simnet group=sys
398 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/smbios group=sys
399 file path=kernel/drv/$(ARCH64)/softmac group=sys
400 file path=kernel/drv/$(ARCH64)/spdsock group=sys
401 file path=kernel/drv/$(ARCH64)/st group=sys
402 file path=kernel/drv/$(ARCH64)/sy group=sys
403 file path=kernel/drv/$(ARCH64)/sysevent group=sys
404 file path=kernel/drv/$(ARCH64)/sysmsg group=sys
405 file path=kernel/drv/$(ARCH64)/tcp group=sys
406 file path=kernel/drv/$(ARCH64)/tcp6 group=sys
407 file path=kernel/drv/$(ARCH64)/tl group=sys
408 $(sparc_ONLY)file path=kernel/drv/$(ARCH64)/ttymux group=sys
409 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/tzmon group=sys
410 $(sparc_ONLY)file path=kernel/drv/$(ARCH64)/uata group=sys
411 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/ucode group=sys
412 file path=kernel/drv/$(ARCH64)/udp group=sys
413 file path=kernel/drv/$(ARCH64)/udp6 group=sys
414 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/vgatext group=sys
415 file path=kernel/drv/$(ARCH64)/vnic group=sys
416 file path=kernel/drv/$(ARCH64)/wc group=sys
417 $(i386_ONLY)file path=kernel/drv/acpi_drv group=sys
418 $(i386_ONLY)file path=kernel/drv/acpi_drv.conf group=sys
419 $(i386_ONLY)file path=kernel/drv/acpi_toshiba group=sys
420 $(i386_ONLY)file path=kernel/drv/aggr group=sys
421 file path=kernel/drv/aggr.conf group=sys
422 $(i386_ONLY)file path=kernel/drv/arp group=sys
423 file path=kernel/drv/arp.conf group=sys
424 $(i386_ONLY)file path=kernel/drv/bl group=sys
425 file path=kernel/drv/bl.conf group=sys
426 $(i386_ONLY)file path=kernel/drv/bridge group=sys
427 file path=kernel/drv/bridge.conf group=sys
428 $(i386_ONLY)file path=kernel/drv/bscbus group=sys
429 $(i386_ONLY)file path=kernel/drv/bscbus.conf group=sys
430 $(i386_ONLY)file path=kernel/drv/bscv group=sys
431 $(i386_ONLY)file path=kernel/drv/bscv.conf group=sys
432 $(i386_ONLY)file path=kernel/drv/clone group=sys
433 file path=kernel/drv/clone.conf group=sys
434 $(i386_ONLY)file path=kernel/drv/cn group=sys
435 file path=kernel/drv/cn.conf group=sys
436 $(i386_ONLY)file path=kernel/drv/conskbd group=sys
437 file path=kernel/drv/conskbd.conf group=sys
438 $(i386_ONLY)file path=kernel/drv/consms group=sys
439 file path=kernel/drv/consms.conf group=sys
440 $(i386_ONLY)file path=kernel/drv/cpuid group=sys
441 file path=kernel/drv/cpuid.conf group=sys
442 $(i386_ONLY)file path=kernel/drv/cpunex group=sys
443 $(i386_ONLY)file path=kernel/drv/crypto group=sys
444 file path=kernel/drv/crypto.conf group=sys
445 $(i386_ONLY)file path=kernel/drv/cryptoadm group=sys
446 file path=kernel/drv/cryptoadm.conf group=sys
447 $(sparc_ONLY)file path=kernel/drv/dad.conf group=sys
448 $(i386_ONLY)file path=kernel/drv/dccp group=sys
449 file path=kernel/drv/dccp.conf group=sys
450 $(i386_ONLY)file path=kernel/drv/dccp6 group=sys
451 file path=kernel/drv/dccp6.conf group=sys
452 #endif /* ! codereview */
453 $(i386_ONLY)file path=kernel/drv/devinfo group=sys
454 file path=kernel/drv/devinfo.conf group=sys
455 $(i386_ONLY)file path=kernel/drv/dld group=sys
456 file path=kernel/drv/dld.conf group=sys
457 $(i386_ONLY)file path=kernel/drv/dlpistub group=sys
```



```
458 file path=kernel/drv/dlpistub.conf group=sys
459 $(i386_ONLY)file path=kernel/drv/i8042 group=sys
460 $(i386_ONLY)file path=kernel/drv/icmp group=sys
461 file path=kernel/drv/icmp.conf group=sys
462 $(i386_ONLY)file path=kernel/drv/icmp6 group=sys
463 file path=kernel/drv/icmp6.conf group=sys
464 $(i386_ONLY)file path=kernel/drv/intel_nb5000 group=sys
465 $(i386_ONLY)file path=kernel/drv/intel_nb5000.conf group=sys
466 $(i386_ONLY)file path=kernel/drv/intel_nhm group=sys
467 $(i386_ONLY)file path=kernel/drv/intel_nhm.conf group=sys
468 $(i386_ONLY)file path=kernel/drv/intel_nhmex group=sys
469 $(i386_ONLY)file path=kernel/drv/intel_nhmex.conf group=sys
470 $(i386_ONLY)file path=kernel/drv/ip group=sys
471 file path=kernel/drv/ip.conf group=sys
472 $(i386_ONLY)file path=kernel/drv/ip6 group=sys
473 file path=kernel/drv/ip6.conf group=sys
474 $(i386_ONLY)file path=kernel/drv/ipnet group=sys
475 file path=kernel/drv/ipnet.conf group=sys
476 $(i386_ONLY)file path=kernel/drv/ippctl group=sys
477 file path=kernel/drv/ippctl.conf group=sys
478 $(i386_ONLY)file path=kernel/drv/ipsecah group=sys
479 file path=kernel/drv/ipsecah.conf group=sys
480 $(i386_ONLY)file path=kernel/drv/ipsecesp group=sys
481 file path=kernel/drv/ipsecesp.conf group=sys
482 $(i386_ONLY)file path=kernel/drv/iptun group=sys
483 file path=kernel/drv/iptun.conf group=sys
484 $(i386_ONLY)file path=kernel/drv/iwscn group=sys
485 file path=kernel/drv/iwscn.conf group=sys
486 $(i386_ONLY)file path=kernel/drv/kb8042 group=sys
487 $(i386_ONLY)file path=kernel/drv/keysock group=sys
488 file path=kernel/drv/keysock.conf group=sys
489 $(i386_ONLY)file path=kernel/drv/kmdb group=sys
490 file path=kernel/drv/kmdb.conf group=sys
491 $(i386_ONLY)file path=kernel/drv/kssl group=sys
492 file path=kernel/drv/kssl.conf group=sys
493 $(i386_ONLY)file path=kernel/drv/llc1 group=sys
494 file path=kernel/drv/llc1.conf group=sys
495 $(i386_ONLY)file path=kernel/drv/lofi group=sys
496 file path=kernel/drv/lofi.conf group=sys
497 $(i386_ONLY)file path=kernel/drv/log group=sys
498 file path=kernel/drv/log.conf group=sys \
499     original_name=SUNWckr:kernel/drv/log.conf preserve=true
500 $(i386_ONLY)file path=kernel/drv/mc-amd group=sys
501 $(i386_ONLY)file path=kernel/drv/mc-amd.conf group=sys
502 $(i386_ONLY)file path=kernel/drv/mm group=sys
503 file path=kernel/drv/mm.conf group=sys
504 $(i386_ONLY)file path=kernel/drv/mouse8042 group=sys
505 $(i386_ONLY)file path=kernel/drv/mpt group=sys
506 $(i386_ONLY)file path=kernel/drv/mpt.conf group=sys \
507     original_name=SUNWckr:kernel/drv/mpt.conf preserve=true
508 $(i386_ONLY)file path=kernel/drv/nulldriver group=sys
509 $(i386_ONLY)file path=kernel/drv/openeepr group=sys
510 file path=kernel/drv/openeepr.conf group=sys
511 $(i386_ONLY)file path=kernel/drv/options group=sys
512 file path=kernel/drv/options.conf group=sys
513 $(i386_ONLY)file path=kernel/drv/pci_pci group=sys
514 $(i386_ONLY)file path=kernel/drv/pcieb group=sys
515 file path=kernel/drv/pcieb.conf group=sys
516 $(i386_ONLY)file path=kernel/drv/physmem group=sys
517 file path=kernel/drv/physmem.conf group=sys
518 $(i386_ONLY)file path=kernel/drv/poll group=sys
519 file path=kernel/drv/poll.conf group=sys
520 $(i386_ONLY)file path=kernel/drv/power group=sys
521 $(i386_ONLY)file path=kernel/drv/power.conf group=sys
522 $(i386_ONLY)file path=kernel/drv/pseudo group=sys
523 file path=kernel/drv/pseudo.conf group=sys
```

```
$(i386_ONLY)file path=kernel/drv/ptc group=sys
file path=kernel/drv/ptc.conf group=sys
$(i386_ONLY)file path=kernel/drv/ptsl group=sys
file path=kernel/drv/ptsl.conf group=sys
$(i386_ONLY)file path=kernel/drv/ramdisk group=sys
file path=kernel/drv/ramdisk.conf group=sys
$(i386_ONLY)file path=kernel/drv/random group=sys
file path=kernel/drv/random.conf group=sys
$(i386_ONLY)file path=kernel/drv/rts group=sys
file path=kernel/drv/rts.conf group=sys
$(i386_ONLY)file path=kernel/drv/sad group=sys
file path=kernel/drv/sad.conf group=sys
$(i386_ONLY)file path=kernel/drv/scsi_vhci group=sys
file path=kernel/drv/scsi_vhci.conf group=sys \
    original_name=SUNWckr:kernel/drv/scsi_vhci.conf preserve=true
$(sparc_ONLY)file path=kernel/drv/sd.conf group=sys \
    original_name=SUNWckr:kernel/drv/sd.conf preserve=true
$(i386_ONLY)file path=kernel/drv/sgen group=sys
file path=kernel/drv/sgen.conf group=sys \
    original_name=SUNWckr:kernel/drv/sgen.conf preserve=true
$(i386_ONLY)file path=kernel/drv/simnet group=sys
file path=kernel/drv/simnet.conf group=sys
$(i386_ONLY)file path=kernel/drv/smbios group=sys
$(i386_ONLY)file path=kernel/drv/smbios.conf group=sys
$(i386_ONLY)file path=kernel/drv/softmac group=sys
file path=kernel/drv/softmac.conf group=sys
$(i386_ONLY)file path=kernel/drv/spdsock group=sys
file path=kernel/drv/spdsock.conf group=sys
$(i386_ONLY)file path=kernel/drv/st group=sys
file path=kernel/drv/st.conf group=sys \
    original_name=SUNWckr:kernel/drv/st.conf preserve=true
$(i386_ONLY)file path=kernel/drv/sy group=sys
file path=kernel/drv/sy.conf group=sys
$(i386_ONLY)file path=kernel/drv/sysevent group=sys
file path=kernel/drv/sysevent.conf group=sys
$(i386_ONLY)file path=kernel/drv/sysmsg group=sys
file path=kernel/drv/sysmsg.conf group=sys
$(i386_ONLY)file path=kernel/drv/tcp group=sys
file path=kernel/drv/tcp.conf group=sys
$(i386_ONLY)file path=kernel/drv/tcp6 group=sys
file path=kernel/drv/tcp6.conf group=sys
$(i386_ONLY)file path=kernel/drv/tl group=sys
file path=kernel/drv/tl.conf group=sys
$(i386_ONLY)file path=kernel/drv/tzmon group=sys
$(i386_ONLY)file path=kernel/drv/tzmon.conf group=sys
$(sparc_ONLY)file path=kernel/drv/uata.conf group=sys \
    original_name=SUNWckr:kernel/drv/uata.conf preserve=true
$(i386_ONLY)file path=kernel/drv/ucode group=sys
$(i386_ONLY)file path=kernel/drv/ucode.conf group=sys
$(i386_ONLY)file path=kernel/drv/udp group=sys
file path=kernel/drv/udp.conf group=sys
$(i386_ONLY)file path=kernel/drv/udp6 group=sys
file path=kernel/drv/udp6.conf group=sys
$(i386_ONLY)file path=kernel/drv/vgatext group=sys
$(i386_ONLY)file path=kernel/drv/vnic group=sys
file path=kernel/drv/vnic.conf group=sys
$(i386_ONLY)file path=kernel/drv/wc group=sys
file path=kernel/drv/wc.conf group=sys
$(sparc_ONLY)file path=kernel/exec/$(ARCH64)/aoutexec group=sys mode=0755
file path=kernel/exec/$(ARCH64)/elfexec group=sys mode=0755
file path=kernel/exec/$(ARCH64)/intpexec group=sys mode=0755
$(i386_ONLY)file path=kernel/exec/elfexec group=sys mode=0755
$(i386_ONLY)file path=kernel/exec/intpexec group=sys mode=0755
file path=kernel/fs/$(ARCH64)/autofs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/cachefs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/ctfs group=sys mode=0755
```



```
file path=kernel/fs/$(ARCH64)/dcfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/dev group=sys mode=0755
file path=kernel/fs/$(ARCH64)/devfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/fifofs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/hsfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/lofs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/mntfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/namefs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/objfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/procfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/sharefs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/sockfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/specfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/tmpfs group=sys mode=0755
file path=kernel/fs/$(ARCH64)/ufs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/autofs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/cachefs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/ctfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/dcfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/dev group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/devfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/fifofs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/hsfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/lofs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/mntfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/namefs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/objfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/procfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/sharefs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/sockfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/specfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/tmpfs group=sys mode=0755
$(i386_ONLY)file path=kernel/fs/ufs group=sys mode=0755
$(i386_ONLY)file path=kernel/genunix group=sys mode=0755
file path=kernel/ipp/$(ARCH64)/ipgpc group=sys mode=0755
$(i386_ONLY)file path=kernel/ipp/ipgpc group=sys mode=0755
file path=kernel/kiconv/$(ARCH64)/kiconv_emea group=sys mode=0755
file path=kernel/kiconv/$(ARCH64)/kiconv_ja group=sys mode=0755
file path=kernel/kiconv/$(ARCH64)/kiconv_ko group=sys mode=0755
file path=kernel/kiconv/$(ARCH64)/kiconv_sc group=sys mode=0755
file path=kernel/kiconv/$(ARCH64)/kiconv_tc group=sys mode=0755
$(i386_ONLY)file path=kernel/kiconv/kiconv_emea group=sys mode=0755
$(i386_ONLY)file path=kernel/kiconv/kiconv_ja group=sys mode=0755
$(i386_ONLY)file path=kernel/kiconv/kiconv_ko group=sys mode=0755
$(i386_ONLY)file path=kernel/kiconv/kiconv_sc group=sys mode=0755
$(i386_ONLY)file path=kernel/kiconv/kiconv_tc group=sys mode=0755
file path=kernel/mac/$(ARCH64)/mac_6to4 group=sys mode=0755
file path=kernel/mac/$(ARCH64)/mac_ether group=sys mode=0755
file path=kernel/mac/$(ARCH64)/mac_ib group=sys mode=0755
file path=kernel/mac/$(ARCH64)/mac_ipv4 group=sys mode=0755
file path=kernel/mac/$(ARCH64)/mac_ipv6 group=sys mode=0755
file path=kernel/mac/$(ARCH64)/mac_wifi group=sys mode=0755
$(i386_ONLY)file path=kernel/mac/mac_6to4 group=sys mode=0755
$(i386_ONLY)file path=kernel/mac/mac_ether group=sys mode=0755
$(i386_ONLY)file path=kernel/mac/mac_ib group=sys mode=0755
$(i386_ONLY)file path=kernel/mac/mac_ipv4 group=sys mode=0755
$(i386_ONLY)file path=kernel/mac/mac_ipv6 group=sys mode=0755
$(i386_ONLY)file path=kernel/mac/mac_wifi group=sys mode=0755
$(i386_ONLY)file path=kernel/misc/$(ARCH64)/acpica group=sys mode=0755
$(i386_ONLY)file path=kernel/misc/$(ARCH64)/agpmaster group=sys mode=0755
file path=kernel/misc/$(ARCH64)/bignum group=sys mode=0755
$(i386_ONLY)file path=kernel/misc/$(ARCH64)/bootdev group=sys mode=0755
file path=kernel/misc/$(ARCH64)/busra group=sys mode=0755
file path=kernel/misc/$(ARCH64)/cardbus group=sys mode=0755
file path=kernel/misc/$(ARCH64)/cmlb group=sys mode=0755
file path=kernel/misc/$(ARCH64)/consconfig group=sys mode=0755
```

```
656 file path=kernel/misc/$(ARCH64)/ctf group=sys mode=0755
657 $(sparc_ONLY)file path=kernel/misc/$(ARCH64)/dada group=sys mode=0755
658 file path=kernel/misc/$(ARCH64)/dls group=sys mode=0755
659 file path=kernel/misc/$(ARCH64)/fssnap_if group=sys mode=0755
660 file path=kernel/misc/$(ARCH64)/gld group=sys mode=0755
661 file path=kernel/misc/$(ARCH64)/hook group=sys mode=0755
662 file path=kernel/misc/$(ARCH64)/hpcsvc group=sys mode=0755
663 file path=kernel/misc/$(ARCH64)/idmap group=sys mode=0755
664 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/iommulib group=sys mode=0755
665 file path=kernel/misc/$(ARCH64)/ipc group=sys mode=0755
666 file path=kernel/misc/$(ARCH64)/kbtrans group=sys mode=0755
667 file path=kernel/misc/$(ARCH64)/kcf group=sys mode=0755
668 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/kmdbmod group=sys mode=0755
669 file path=kernel/misc/$(ARCH64)/ksocket group=sys mode=0755
670 file path=kernel/misc/$(ARCH64)/mac group=sys mode=0755
671 file path=kernel/misc/$(ARCH64)/mii group=sys mode=0755
672 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/net80211 group=sys mode=0755
673 file path=kernel/misc/$(ARCH64)/neti group=sys mode=0755
674 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/pci_autoconfig group=sys mode=0755
675 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/pcicfg group=sys mode=0755
676 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/pcie group=sys mode=0755
677 file path=kernel/misc/$(ARCH64)/pcihp group=sys mode=0755
678 file path=kernel/misc/$(ARCH64)/pcmcia group=sys mode=0755
679 file path=kernel/misc/$(ARCH64)/rpcsec group=sys mode=0755
680 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/sata group=sys mode=0755
681 file path=kernel/misc/$(ARCH64)/scsi group=sys mode=0755
682 file path=kernel/misc/$(ARCH64)/strplumb group=sys mode=0755
683 $(sparc_ONLY)file path=kernel/misc/$(ARCH64)/swapgeneric group=sys mode=0755
684 file path=kernel/misc/$(ARCH64)/tem group=sys mode=0755
685 file path=kernel/misc/$(ARCH64)/tlimod group=sys mode=0755
686 $(i386_ONLY)file path=kernel/misc/acpica group=sys mode=0755
687 $(i386_ONLY)file path=kernel/misc/agpmaster group=sys mode=0755
688 $(i386_ONLY)file path=kernel/misc/bignum group=sys mode=0755
689 $(i386_ONLY)file path=kernel/misc/bootdev group=sys mode=0755
690 $(i386_ONLY)file path=kernel/misc/busra group=sys mode=0755
691 $(i386_ONLY)file path=kernel/misc/cardbus group=sys mode=0755
692 $(i386_ONLY)file path=kernel/misc/cmlb group=sys mode=0755
693 $(i386_ONLY)file path=kernel/misc/consconfig group=sys mode=0755
694 $(i386_ONLY)file path=kernel/misc/ctf group=sys mode=0755
695 $(i386_ONLY)file path=kernel/misc/dls group=sys mode=0755
696 $(i386_ONLY)file path=kernel/misc/fssnap_if group=sys mode=0755
697 $(i386_ONLY)file path=kernel/misc/gld group=sys mode=0755
698 $(i386_ONLY)file path=kernel/misc/hook group=sys mode=0755
699 $(i386_ONLY)file path=kernel/misc/hpcsvc group=sys mode=0755
700 $(i386_ONLY)file path=kernel/misc/idmap group=sys mode=0755
701 $(i386_ONLY)file path=kernel/misc/iommulib group=sys mode=0755
702 $(i386_ONLY)file path=kernel/misc/ipc group=sys mode=0755
703 $(i386_ONLY)file path=kernel/misc/kbtrans group=sys mode=0755
704 $(i386_ONLY)file path=kernel/misc/kcf group=sys mode=0755
705 $(i386_ONLY)file path=kernel/misc/kmdbmod group=sys mode=0755
706 $(i386_ONLY)file path=kernel/misc/ksocket group=sys mode=0755
707 $(i386_ONLY)file path=kernel/misc/mac group=sys mode=0755
708 $(i386_ONLY)file path=kernel/misc/mii group=sys mode=0755
709 $(i386_ONLY)file path=kernel/misc/net80211 group=sys mode=0755
710 $(i386_ONLY)file path=kernel/misc/neti group=sys mode=0755
711 $(i386_ONLY)file path=kernel/misc/pci_autoconfig group=sys mode=0755
712 $(i386_ONLY)file path=kernel/misc/pcicfg group=sys mode=0755
713 $(i386_ONLY)file path=kernel/misc/pcie group=sys mode=0755
714 $(i386_ONLY)file path=kernel/misc/pcihp group=sys mode=0755
715 $(i386_ONLY)file path=kernel/misc/pcmcia group=sys mode=0755
716 $(i386_ONLY)file path=kernel/misc/rpcsec group=sys mode=0755
717 $(i386_ONLY)file path=kernel/misc/sata group=sys mode=0755
718 $(i386_ONLY)file path=kernel/misc/scsi group=sys mode=0755
719 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_asym_emc group=sys \
720     mode=0755
721 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_asym_lsi group=sys \
```



```
722     mode=0755
723 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_asym_sun group=sys \
724     mode=0755
725 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_sym group=sys mode=0755
726 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_sym_emc group=sys \
727     mode=0755
728 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_sym_hds group=sys \
729     mode=0755
730 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_tape group=sys mode=0755
731 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_tpgs group=sys mode=0755
732 file path=kernel/misc/scsi_vhci/$(ARCH64)/scsi_vhci_f_tpgs_tape group=sys \
733     mode=0755
734 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_asym_emc group=sys \
735     mode=0755
736 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_asym_lsi group=sys \
737     mode=0755
738 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_asym_sun group=sys \
739     mode=0755
740 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_sym group=sys \
741     mode=0755
742 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_sym_emc group=sys \
743     mode=0755
744 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_sym_hds group=sys \
745     mode=0755
746 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_tape group=sys \
747     mode=0755
748 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_tpgs group=sys \
749     mode=0755
750 $(i386_ONLY)file path=kernel/misc/scsi_vhci/scsi_vhci_f_tpgs_tape group=sys \
751     mode=0755
752 $(i386_ONLY)file path=kernel/misc/strplumb group=sys mode=0755
753 $(i386_ONLY)file path=kernel/misc/tem group=sys mode=0755
754 $(i386_ONLY)file path=kernel/misc/tlimod group=sys mode=0755
755 file path=kernel/sched/$(ARCH64)/SDC group=sys mode=0755
756 file path=kernel/sched/$(ARCH64)/TS group=sys mode=0755
757 file path=kernel/sched/$(ARCH64)/TS_DPTBL group=sys mode=0755
758 $(i386_ONLY)file path=kernel/sched/SDC group=sys mode=0755
759 $(i386_ONLY)file path=kernel/sched/TS group=sys mode=0755
760 $(i386_ONLY)file path=kernel/sched/TS_DPTBL group=sys mode=0755
761 file path=kernel/socketmod/$(ARCH64)/ksslf group=sys mode=0755
762 file path=kernel/socketmod/$(ARCH64)/socksctp group=sys mode=0755
763 file path=kernel/socketmod/$(ARCH64)/trill group=sys mode=0755
764 $(i386_ONLY)file path=kernel/socketmod/ksslf group=sys mode=0755
765 $(i386_ONLY)file path=kernel/socketmod/socksctp group=sys mode=0755
766 $(i386_ONLY)file path=kernel/socketmod/trill group=sys mode=0755
767 file path=kernel/strmod/$(ARCH64)/bufmod group=sys mode=0755
768 file path=kernel/strmod/$(ARCH64)/connld group=sys mode=0755
769 file path=kernel/strmod/$(ARCH64)/dedump group=sys mode=0755
770 file path=kernel/strmod/$(ARCH64)/drcompat group=sys mode=0755
771 file path=kernel/strmod/$(ARCH64)/ldterm group=sys mode=0755
772 $(sparc_ONLY)file path=kernel/strmod/$(ARCH64)/ms group=sys mode=0755
773 file path=kernel/strmod/$(ARCH64)/pckt group=sys mode=0755
774 file path=kernel/strmod/$(ARCH64)/pfmod group=sys mode=0755
775 file path=kernel/strmod/$(ARCH64)/pipemod group=sys mode=0755
776 file path=kernel/strmod/$(ARCH64)/ptem group=sys mode=0755
777 file path=kernel/strmod/$(ARCH64)/redirmod group=sys mode=0755
778 file path=kernel/strmod/$(ARCH64)/rpcmod group=sys mode=0755
779 file path=kernel/strmod/$(ARCH64)/timod group=sys mode=0755
780 file path=kernel/strmod/$(ARCH64)/tirdwr group=sys mode=0755
781 file path=kernel/strmod/$(ARCH64)/ttcompat group=sys mode=0755
782 $(sparc_ONLY)file path=kernel/strmod/$(ARCH64)/vuid3ps2 group=sys mode=0755
783 $(i386_ONLY)file path=kernel/strmod/bufmod group=sys mode=0755
784 $(i386_ONLY)file path=kernel/strmod/connld group=sys mode=0755
785 $(i386_ONLY)file path=kernel/strmod/dedump group=sys mode=0755
786 $(i386_ONLY)file path=kernel/strmod/drcompat group=sys mode=0755
787 $(i386_ONLY)file path=kernel/strmod/ldterm group=sys mode=0755
```

```
$(i386_ONLY)file path=kernel/strmod/pckt group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/pfmod group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/pipemod group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/ptem group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/redirmod group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/rpcmod group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/timod group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/tirdwr group=sys mode=0755
$(i386_ONLY)file path=kernel/strmod/ttcompat group=sys mode=0755
file path=kernel/sys/$(ARCH64)/c2audit group=sys mode=0755
file path=kernel/sys/$(ARCH64)/doorfs group=sys mode=0755
file path=kernel/sys/$(ARCH64)/inst_sync group=sys mode=0755
file path=kernel/sys/$(ARCH64)/kaio group=sys mode=0755
file path=kernel/sys/$(ARCH64)/msgsys group=sys mode=0755
file path=kernel/sys/$(ARCH64)/pipe group=sys mode=0755
file path=kernel/sys/$(ARCH64)/portfs group=sys mode=0755
file path=kernel/sys/$(ARCH64)/pset group=sys mode=0755
file path=kernel/sys/$(ARCH64)/semsys group=sys mode=0755
file path=kernel/sys/$(ARCH64)/shmsys group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/c2audit group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/doorfs group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/inst_sync group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/kaio group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/msgsys group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/pipe group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/portfs group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/pset group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/semsys group=sys mode=0755
$(i386_ONLY)file path=kernel/sys/shmsys group=sys mode=0755
file path=lib/svc/manifest/system/dumpadm.xml group=sys mode=0444
file path=lib/svc/manifest/system/intrd.xml group=sys mode=0444
file path=lib/svc/manifest/system/scheduler.xml group=sys mode=0444
file path=lib/svc/method/svc-dumpadm mode=0555
file path=lib/svc/method/svc-intrd mode=0555
file path=lib/svc/method/svc-scheduler mode=0555
$(sparc_ONLY)file path=usr/share/man/man1m/monitor.1m
$(sparc_ONLY)file path=usr/share/man/man1m/obpsym.1m
# On SPARC driver/bscv is Serverblade1 specific, and in system/kernel/platform
# We keep the manual page generic
$(sparc_ONLY)file path=usr/share/man/man7d/dad.7d
$(i386_ONLY)file path=usr/share/man/man7d/smbios.7d
# Sadly vuid mouse support is in different packages on different platforms
# While kstat(7D) is in SUNWcs, the structures are general
hardlink path=kernel/misc/$(ARCH64)/md5 \
    target=../../../kernel/crypto/$(ARCH64)/md5
hardlink path=kernel/misc/$(ARCH64)/sha1 \
    target=../../../kernel/crypto/$(ARCH64)/sha1
hardlink path=kernel/misc/$(ARCH64)/sha2 \
    target=../../../kernel/crypto/$(ARCH64)/sha2
$(i386_ONLY)hardlink path=kernel/misc/md5 target=../../kernel/crypto/md5
$(i386_ONLY)hardlink path=kernel/misc/sha1 target=../../kernel/crypto/sha1
$(i386_ONLY)hardlink path=kernel/misc/sha2 target=../../kernel/crypto/sha2
hardlink path=kernel/socketmod/$(ARCH64)/dccp \
    target=../../../kernel/drv/$(ARCH64)/dccp
#endif /* ! codereview */
hardlink path=kernel/socketmod/$(ARCH64)/icmp \
    target=../../../kernel/drv/$(ARCH64)/icmp
hardlink path=kernel/socketmod/$(ARCH64)/rts \
    target=../../../kernel/drv/$(ARCH64)/rts
hardlink path=kernel/socketmod/$(ARCH64)/tcp \
    target=../../../kernel/drv/$(ARCH64)/tcp
hardlink path=kernel/socketmod/$(ARCH64)/udp \
    target=../../../kernel/drv/$(ARCH64)/udp
$(i386_ONLY)hardlink path=kernel/socketmod/dccp target=../../kernel/drv/dccp
#endif /* ! codereview */
$(i386_ONLY)hardlink path=kernel/socketmod/icmp target=../../kernel/drv/icmp
```



```
$(i386_ONLY)hardlink path=kernel/socketmod/rts target=../../kernel/drv/rts
$(i386_ONLY)hardlink path=kernel/socketmod/tcp target=../../kernel/drv/tcp
$(i386_ONLY)hardlink path=kernel/socketmod/udp target=../../kernel/drv/udp
hardlink path=kernel/strmod/$(ARCH64)/arp \
    target=../../../kernel/drv/$(ARCH64)/arp
hardlink path=kernel/strmod/$(ARCH64)/dccp \
    target=../../../kernel/drv/$(ARCH64)/dccp
#endif /* ! codereview */
hardlink path=kernel/strmod/$(ARCH64)/icmp \
    target=../../../kernel/drv/$(ARCH64)/icmp
hardlink path=kernel/strmod/$(ARCH64)/ip \
    target=../../../kernel/drv/$(ARCH64)/ip
hardlink path=kernel/strmod/$(ARCH64)/ipsecah \
    target=../../../kernel/drv/$(ARCH64)/ipsecah
hardlink path=kernel/strmod/$(ARCH64)/ipsecesp \
    target=../../../kernel/drv/$(ARCH64)/ipsecesp
hardlink path=kernel/strmod/$(ARCH64)/keysock \
    target=../../../kernel/drv/$(ARCH64)/keysock
hardlink path=kernel/strmod/$(ARCH64)/tcp \
    target=../../../kernel/drv/$(ARCH64)/tcp
hardlink path=kernel/strmod/$(ARCH64)/udp \
    target=../../../kernel/drv/$(ARCH64)/udp
$(i386_ONLY)hardlink path=kernel/strmod/arp target=../../kernel/drv/arp
$(i386_ONLY)hardlink path=kernel/strmod/dccp target=../../kernel/drv/dccp
#endif /* ! codereview */
$(i386_ONLY)hardlink path=kernel/strmod/icmp target=../../kernel/drv/icmp
$(i386_ONLY)hardlink path=kernel/strmod/ip target=../../kernel/drv/ip
$(i386_ONLY)hardlink path=kernel/strmod/ipsecah \
    target=../../kernel/drv/ipsecah
$(i386_ONLY)hardlink path=kernel/strmod/ipsecesp \
    target=../../kernel/drv/ipsecesp
$(i386_ONLY)hardlink path=kernel/strmod/keysock \
    target=../../kernel/drv/keysock
$(i386_ONLY)hardlink path=kernel/strmod/tcp target=../../kernel/drv/tcp
$(i386_ONLY)hardlink path=kernel/strmod/udp target=../../kernel/drv/udp
hardlink path=kernel/sys/$(ARCH64)/autofs \
    target=../../../kernel/fs/$(ARCH64)/autofs
hardlink path=kernel/sys/$(ARCH64)/rpcmod \
    target=../../../kernel/strmod/$(ARCH64)/rpcmod
$(i386_ONLY)hardlink path=kernel/sys/autofs target=../../kernel/fs/autofs
$(i386_ONLY)hardlink path=kernel/sys/rpcmod target=../../kernel/strmod/rpcmod
legacy pkg=SUNWckr \
    desc="core kernel software for a specific instruction-set architecture" \
    name="Core Solaris Kernel (Root)"
license cr_Sun license=cr_Sun
license lic_CDDL license=lic_CDDL
license usr/src/cmd/mdb/common/libstand/THIRDPARTYLICENSE \
    license=usr/src/cmd/mdb/common/libstand/THIRDPARTYLICENSE
license usr/src/common/bzip2/LICENSE license=usr/src/common/bzip2/LICENSE
license usr/src/common/crypto/THIRDPARTYLICENSE.cryptogams \
    license=usr/src/common/crypto/THIRDPARTYLICENSE.cryptogams
$(i386_ONLY)license usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.gladman \
    license=usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.gladman
$(i386_ONLY)license usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.openssl \
    license=usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.openssl
license usr/src/common/crypto/ecc/THIRDPARTYLICENSE \
    license=usr/src/common/crypto/ecc/THIRDPARTYLICENSE
$(i386_ONLY)license usr/src/common/crypto/md5/amd64/THIRDPARTYLICENSE \
    license=usr/src/common/crypto/md5/amd64/THIRDPARTYLICENSE
license usr/src/common/mpi/THIRDPARTYLICENSE \
    license=usr/src/common/mpi/THIRDPARTYLICENSE
license usr/src/uts/common/inet/ip/THIRDPARTYLICENSE.rts \
    license=usr/src/uts/common/inet/ip/THIRDPARTYLICENSE.rts
license usr/src/uts/common/inet/tcp/THIRDPARTYLICENSE \
    license=usr/src/uts/common/inet/tcp/THIRDPARTYLICENSE
license usr/src/uts/common/io/THIRDPARTYLICENSE.etheraddr \
```

```
920     license=usr/src/uts/common/io/THIRDPARTYLICENSE.etheraddr
921 license usr/src/uts/common/sys/THIRDPARTYLICENSE.icu \
922     license=usr/src/uts/common/sys/THIRDPARTYLICENSE.icu
923 license usr/src/uts/common/sys/THIRDPARTYLICENSE.unicode \
924     license=usr/src/uts/common/sys/THIRDPARTYLICENSE.unicode
925 $(i386_ONLY)license usr/src/uts/intel/io/acpica/THIRDPARTYLICENSE \
926     license=usr/src/uts/intel/io/acpica/THIRDPARTYLICENSE
927 $(i386_ONLY)link path=boot/solaris/bin/root_archive \
928     target=../../../usr/sbin/root_archive
929 link path=dev/dld target=../devices/pseudo/dld@0:ctl
930 link path=kernel/misc/$(ARCH64)/des \
931     target=../../../kernel/crypto/$(ARCH64)/des
932 $(i386_ONLY)link path=kernel/misc/des target=../../kernel/crypto/des
```

```
**********************************************************
   43063 Mon Jul  9 14:38:11 2012
new/usr/src/uts/common/Makefile.files
dccp: starting module template
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#

#
# This Makefile defines all file modules for the directory uts/common
# and its children. These are the source files which may be considered
# common to all SunOS systems.

i386_CORE_OBJS += \
                atomic.o        \
                avintr.o        \
                pic.o

sparc_CORE_OBJS +=

COMMON_CORE_OBJS +=             \
                beep.o          \
                bitset.o        \
                bp_map.o        \
                brand.o         \
                cpucaps.o       \
                cmt.o           \
                cmt_policy.o    \
                cpu.o           \
                cpu_event.o     \
                cpu_intr.o      \
                cpu_pm.o        \
                cpupart.o       \
                cap_util.o      \
                disp.o          \
                group.o         \
                kstat_fr.o      \
                iscsiboot_prop.o        \
                lgrp.o          \
                lgrp_topo.o     \
                mmapobj.o       \
                mutex.o         \
```



```
                page_lock.o     \
                page_retire.o   \
                panic.o         \
                param.o         \
                pg.o            \
                pghw.o          \
                putnext.o       \
                rctl_proc.o     \
                rwlock.o        \
                seg_kmem.o      \
                softint.o       \
                string.o        \
                strtol.o        \
                strtoul.o       \
                strtoll.o       \
                strtoull.o      \
                thread_intr.o   \
                vm_page.o       \
                vm_pagelist.o   \
                zlib_obj.o      \
                clock_tick.o

CORE_OBJS +=    $(COMMON_CORE_OBJS) $($(MACH)_CORE_OBJS)

ZLIB_OBJS =     zutil.o zmod.o zmod_subr.o \
                adler32.o crc32.o deflate.o inffast.o \
                inflate.o inftrees.o trees.o

GENUNIX_OBJS += \
                access.o        \
                acl.o           \
                acl_common.o    \
                adjtime.o       \
                alarm.o         \
                aio_subr.o      \
                auditsys.o      \
                audit_core.o    \
                audit_zone.o    \
                audit_memory.o  \
                autoconf.o      \
                avl.o           \
                bdev_dsort.o    \
                bio.o           \
                bitmap.o        \
                blabel.o        \
                brandsys.o      \
                bz2blocksort.o  \
                bz2compress.o   \
                bz2decompress.o \
                bz2randtable.o  \
                bz2bzlib.o      \
                bz2crctable.o   \
                bz2huffman.o    \
                callb.o         \
                callout.o       \
                chdir.o         \
                chmod.o         \
                chown.o         \
                cladm.o         \
                class.o         \
                clock.o         \
                clock_highres.o \
                clock_realtime.o\
                close.o         \
                compress.o      \
                condvar.o       \
```

```
128                 conf.o          \
129                 console.o       \
130                 contract.o      \
131                 copyops.o       \
132                 core.o          \
133                 corectl.o       \
134                 cred.o          \
135                 cs_stubs.o      \
136                 dacf.o          \
137                 dacf_clnt.o     \
138                 damap.o \
139                 cyclic.o        \
140                 ddi.o           \
141                 ddifm.o         \
142                 ddi_hp_impl.o   \
143                 ddi_hp_ndi.o    \
144                 ddi_intr.o      \
145                 ddi_intr_impl.o \
146                 ddi_intr_irm.o  \
147                 ddi_nodeid.o    \
148                 ddi_timer.o     \
149                 devcfg.o        \
150                 devcache.o      \
151                 device.o        \
152                 devid.o         \
153                 devid_cache.o   \
154                 devid_scsi.o    \
155                 devid_smp.o     \
156                 devpolicy.o     \
157                 disp_lock.o     \
158                 dnlc.o          \
159                 driver.o        \
160                 dumpsubr.o      \
161                 driver_lyr.o    \
162                 dtrace_subr.o   \
163                 errorq.o        \
164                 etheraddr.o     \
165                 evchannels.o    \
166                 exacct.o        \
167                 exacct_core.o   \
168                 exec.o          \
169                 exit.o          \
170                 fbio.o          \
171                 fcntl.o         \
172                 fdbuffer.o      \
173                 fdsync.o        \
174                 fem.o           \
175                 ffs.o           \
176                 fio.o           \
177                 flock.o         \
178                 fm.o            \
179                 fork.o          \
180                 vpm.o           \
181                 fs_reparse.o    \
182                 fs_subr.o       \
183                 fsflush.o       \
184                 ftrace.o        \
185                 getcwd.o        \
186                 getdents.o      \
187                 getloadavg.o    \
188                 getpagesizes.o  \
189                 getpid.o        \
190                 gfs.o           \
191                 rusagesys.o     \
192                 gid.o           \
193                 groups.o        \
```



```
194                 grow.o          \
195                 hat_refmod.o    \
196                 id32.o          \
197                 id_space.o      \
198                 inet_ntop.o     \
199                 instance.o      \
200                 ioctl.o         \
201                 ip_cksum.o      \
202                 issetugid.o     \
203                 ippconf.o       \
204                 kcpc.o          \
205                 kdi.o           \
206                 kiconv.o        \
207                 klpd.o          \
208                 kmem.o          \
209                 ksyms_snapshot.o        \
210                 l_strplumb.o    \
211                 labelsys.o      \
212                 link.o          \
213                 list.o          \
214                 lockstat_subr.o \
215                 log_sysevent.o  \
216                 logsubr.o       \
217                 lookup.o        \
218                 lseek.o         \
219                 ltos.o          \
220                 lwp.o           \
221                 lwp_create.o    \
222                 lwp_info.o      \
223                 lwp_self.o      \
224                 lwp_sobj.o      \
225                 lwp_timer.o     \
226                 lwpsys.o        \
227                 main.o          \
228                 mmapobjsys.o    \
229                 memcntl.o       \
230                 memstr.o        \
231                 lgrpsys.o       \
232                 mkdir.o         \
233                 mknod.o         \
234                 mount.o         \
235                 move.o          \
236                 msacct.o        \
237                 multidata.o     \
238                 nbmlock.o       \
239                 ndifm.o         \
240                 nice.o          \
241                 netstack.o      \
242                 ntptime.o       \
243                 nvpair.o        \
244                 nvpair_alloc_system.o   \
245                 nvpair_alloc_fixed.o    \
246                 fnvpair.o       \
247                 octet.o         \
248                 open.o          \
249                 p_online.o      \
250                 pathconf.o      \
251                 pathname.o      \
252                 pause.o         \
253                 serializer.o    \
254                 pci_intr_lib.o  \
255                 pci_cap.o       \
256                 pcifm.o         \
257                 pgrp.o          \
258                 pgrpsys.o       \
259                 pid.o           \
```

```
260                 pkp_hash.o      \
261                 policy.o        \
262                 poll.o          \
263                 pool.o          \
264                 pool_pset.o     \
265                 port_subr.o     \
266                 ppriv.o         \
267                 printf.o        \
268                 priocntl.o      \
269                 priv.o          \
270                 priv_const.o    \
271                 proc.o          \
272                 procset.o       \
273                 processor_bind.o        \
274                 processor_info.o        \
275                 profil.o        \
276                 project.o       \
277                 qsort.o         \
278                 rctl.o          \
279                 rctlsys.o       \
280                 readlink.o      \
281                 refstr.o        \
282                 rename.o        \
283                 resolvepath.o   \
284                 retire_store.o  \
285                 process.o       \
286                 rlimit.o        \
287                 rmap.o          \
288                 rw.o            \
289                 rwstlock.o      \
290                 sad_conf.o      \
291                 sid.o           \
292                 sidsys.o        \
293                 sched.o         \
294                 schedctl.o      \
295                 sctp_crc32.o    \
296                 seg_dev.o       \
297                 seg_kp.o        \
298                 seg_kpm.o       \
299                 seg_map.o       \
300                 seg_vn.o        \
301                 seg_spt.o       \
302                 semaphore.o     \
303                 sendfile.o      \
304                 session.o       \
305                 share.o         \
306                 shuttle.o       \
307                 sig.o           \
308                 sigaction.o     \
309                 sigaltstack.o   \
310                 signotify.o     \
311                 sigpending.o    \
312                 sigprocmask.o   \
313                 sigqueue.o      \
314                 sigsendset.o    \
315                 sigsuspend.o    \
316                 sigtimedwait.o  \
317                 sleepq.o        \
318                 sock_conf.o     \
319                 space.o         \
320                 sscanf.o        \
321                 stat.o          \
322                 statfs.o        \
323                 statvfs.o       \
324                 stol.o          \
325                 str_conf.o      \
```



```
326                 strcalls.o      \
327                 stream.o        \
328                 streamio.o      \
329                 strext.o        \
330                 strsubr.o       \
331                 strsun.o        \
332                 subr.o          \
333                 sunddi.o        \
334                 sunmdi.o        \
335                 sunndi.o        \
336                 sunpci.o        \
337                 sunpm.o         \
338                 sundlpi.o       \
339                 suntpi.o        \
340                 swap_subr.o     \
341                 swap_vnops.o    \
342                 symlink.o       \
343                 sync.o          \
344                 sysclass.o      \
345                 sysconfig.o     \
346                 sysent.o        \
347                 sysfs.o         \
348                 systeminfo.o    \
349                 task.o          \
350                 taskq.o         \
351                 tasksys.o       \
352                 time.o          \
353                 timer.o         \
354                 times.o         \
355                 timers.o        \
356                 thread.o        \
357                 tlabel.o        \
358                 tnf_res.o       \
359                 turnstile.o     \
360                 tty_common.o    \
361                 u8_textprep.o   \
362                 uadmin.o        \
363                 uconv.o         \
364                 ucredsys.o      \
365                 uid.o           \
366                 umask.o         \
367                 umount.o        \
368                 uname.o         \
369                 unix_bb.o       \
370                 unlink.o        \
371                 urw.o           \
372                 utime.o         \
373                 utssys.o        \
374                 uucopy.o        \
375                 vfs.o           \
376                 vfs_conf.o      \
377                 vmem.o          \
378                 vm_anon.o       \
379                 vm_as.o         \
380                 vm_meter.o      \
381                 vm_pageout.o    \
382                 vm_pvn.o        \
383                 vm_rm.o         \
384                 vm_seg.o        \
385                 vm_subr.o       \
386                 vm_swap.o       \
387                 vm_usage.o      \
388                 vnode.o         \
389                 vuid_queue.o    \
390                 vuid_store.o    \
391                 waitq.o         \
```

```
392                 watchpoint.o    \
393                 yield.o         \
394                 scsi_confdata.o \
395                 xattr.o         \
396                 xattr_common.o  \
397                 xdr_mblk.o      \
398                 xdr_mem.o       \
399                 xdr.o           \
400                 xdr_array.o     \
401                 xdr_refer.o     \
402                 xhat.o          \
403                 zone.o

405 #
406 #       Stubs for the stand-alone linker/loader
407 #
408 sparc_GENSTUBS_OBJS =   \
409         kobj_stubs.o

411 i386_GENSTUBS_OBJS =

413 COMMON_GENSTUBS_OBJS =

415 GENSTUBS_OBJS += $(COMMON_GENSTUBS_OBJS) $($(MACH)_GENSTUBS_OBJS)

417 #
418 #       DTrace and DTrace Providers
419 #
420 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

422 SDT_OBJS += sdt_subr.o

424 PROFILE_OBJS += profile.o

426 SYSTRACE_OBJS += systrace.o

428 LOCKSTAT_OBJS += lockstat.o

430 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

432 DCPC_OBJS += dcpc.o

434 #
435 #       Driver (pseudo-driver) Modules
436 #
437 IPP_OBJS +=     ippctl.o

439 AUDIO_OBJS += audio_client.o audio_ddi.o audio_engine.o \
440             audio_fltdata.o audio_format.o audio_ctrl.o \
441             audio_grc3.o audio_output.o audio_input.o \
442             audio_oss.o audio_sun.o

444 AUDIOEMU10K_OBJS += audioemu10k.o

446 AUDIOENS_OBJS += audioens.o

448 AUDIOVIA823X_OBJS += audiovia823x.o

450 AUDIOVIA97_OBJS += audiovia97.o

452 AUDIO1575_OBJS += audio1575.o

454 AUDIO810_OBJS += audio810.o

456 AUDIOCMI_OBJS += audiocmi.o
```



```
458 AUDIOCMIHD_OBJS += audiocmihd.o

460 AUDIOHD_OBJS += audiohd.o

462 AUDIOIXP_OBJS += audioixp.o

464 AUDIOLS_OBJS += audiols.o

466 AUDIOP16X_OBJS += audiop16x.o

468 AUDIOPCI_OBJS += audiopci.o

470 AUDIOSOLO_OBJS += audiosolo.o

472 AUDIOTS_OBJS += audiots.o

474 AC97_OBJS += ac97.o ac97_ad.o ac97_alc.o ac97_cmi.o

476 BLKDEV_OBJS += blkdev.o

478 CARDBUS_OBJS += cardbus.o cardbus_hp.o cardbus_cfg.o

480 CONSKBD_OBJS += conskbd.o

482 CONSMS_OBJS +=  consms.o

484 OLDPTY_OBJS +=  tty_ptyconf.o

486 PTC_OBJS +=     tty_pty.o

488 PTSL_OBJS +=    tty_pts.o

490 PTM_OBJS +=     ptm.o

492 MII_OBJS +=     mii.o mii_cicada.o mii_natsemi.o mii_intel.o mii_qualsemi.o \
493                 mii_marvell.o mii_realtek.o mii_other.o

495 PTS_OBJS +=     pts.o

497 PTY_OBJS +=     ptms_conf.o

499 SAD_OBJS +=     sad.o

501 MD4_OBJS +=     md4.o md4_mod.o

503 MD5_OBJS +=     md5.o md5_mod.o

505 SHA1_OBJS +=    sha1.o sha1_mod.o

507 SHA2_OBJS +=    sha2.o sha2_mod.o

509 IPGPC_OBJS +=   classifierddi.o classifier.o filters.o trie.o table.o \
510                 ba_table.o

512 DSCPMK_OBJS +=  dscpmk.o dscpmkddi.o

514 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o

516 FLOWACCT_OBJS +=        flowacctddi.o flowacct.o

518 TOKENMT_OBJS += tokenmt.o tokenmtddi.o

520 TSWTCL_OBJS +=  tswtcl.o tswtclddi.o

522 ARP_OBJS +=     arpddi.o
```

```
524 ICMP_OBJS +=    icmpddi.o

526 ICMP6_OBJS +=   icmp6ddi.o

528 RTS_OBJS +=     rtsddi.o

530 IP_ICMP_OBJS =  icmp.o icmp_opt_data.o
531 IP_RTS_OBJS =   rts.o rts_opt_data.o
532 IP_TCP_OBJS =   tcp.o tcp_fusion.o tcp_opt_data.o tcp_sack.o tcp_stats.o \
533                 tcp_misc.o tcp_timers.o tcp_time_wait.o tcp_tpi.o tcp_output.o \
534                 tcp_input.o tcp_socket.o tcp_bind.o tcp_cluster.o tcp_tunables.o
535 IP_UDP_OBJS =   udp.o udp_opt_data.o udp_tunables.o udp_stats.o
536 IP_SCTP_OBJS =  sctp.o sctp_opt_data.o sctp_output.o \
537                 sctp_init.o sctp_input.o sctp_cookie.o \
538                 sctp_conn.o sctp_error.o sctp_snmp.o \
539                 sctp_tunables.o sctp_shutdown.o sctp_common.o \
540                 sctp_timer.o sctp_heartbeat.o sctp_hash.o \
541                 sctp_bind.o sctp_notify.o sctp_asconf.o \
542                 sctp_addr.o tn_ipopt.o tnet.o ip_netinfo.o \
543                 sctp_misc.o
544 IP_ILB_OBJS =   ilb.o ilb_nat.o ilb_conn.o ilb_alg_hash.o ilb_alg_rr.o
545 IP_DCCP_OBJS =  dccp.o dccp_bind.o dccp_features.o dccp_input.o dccp_misc.o \
546                 dccp_opt_data.o dccp_output.o dccp_stats.o dccp_socket.o \
547                 dccp_tpi.o dccp_tunables.o
548 #endif /* ! codereview */

550 IP_OBJS +=      igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
551                 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
552                 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
553                 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
554                 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_squeue.o \
555                 squeue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
556                 ip_helper_stream.o ip_tunables.o \
557                 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
558                 conn_opt.o ip_attr.o ip_dce.o \
559                 $(IP_ICMP_OBJS) \
560                 $(IP_RTS_OBJS) \
561                 $(IP_TCP_OBJS) \
562                 $(IP_UDP_OBJS) \
563                 $(IP_SCTP_OBJS) \
564                 $(IP_ILB_OBJS) \
565                 $(IP_DCCP_OBJS)
545                 $(IP_ILB_OBJS)

567 IP6_OBJS +=     ip6ddi.o

569 HOOK_OBJS +=    hook.o

571 NETI_OBJS +=    neti_impl.o neti_mod.o neti_stack.o

573 KEYSOCK_OBJS += keysockddi.o keysock.o keysock_opt_data.o

575 IPNET_OBJS +=   ipnet.o ipnet_bpf.o

577 SPDSOCK_OBJS += spdsockddi.o spdsock.o spdsock_opt_data.o

579 IPSECESP_OBJS += ipsecespddi.o ipsecesp.o

581 IPSECAH_OBJS += ipsecahddi.o ipsecah.o sadb.o

583 SPPP_OBJS +=    sppp.o sppp_dlpi.o sppp_mod.o s_common.o

585 SPPPTUN_OBJS += sppptun.o sppptun_mod.o

587 SPPPASYN_OBJS += spppasyn.o spppasyn_mod.o
```



```
589 SPPPCOMP_OBJS += spppcomp.o spppcomp_mod.o deflate.o bsd-comp.o vjcompress.o \
590                 zlib.o

592 TCP_OBJS +=     tcpddi.o

594 TCP6_OBJS +=    tcp6ddi.o

596 NCA_OBJS +=     ncaddi.o

598 SDP_SOCK_MOD_OBJS += sockmod_sdp.o socksdp.o socksdpsubr.o

600 SCTP_SOCK_MOD_OBJS += sockmod_sctp.o socksctp.o socksctpsubr.o

602 PFP_SOCK_MOD_OBJS += sockmod_pfp.o

604 RDS_SOCK_MOD_OBJS += sockmod_rds.o

606 RDS_OBJS +=     rdsddi.o rdssubr.o rds_opt.o rds_ioctl.o

608 RDSIB_OBJS +=   rdsib.o rdsib_ib.o rdsib_cm.o rdsib_ep.o rdsib_buf.o \
609                 rdsib_debug.o rdsib_sc.o

611 RDSV3_OBJS +=   af_rds.o rdsv3_ddi.o bind.o loop.o threads.o connection.o \
612                 transport.o cong.o sysctl.o message.o rds_recv.o send.o \
613                 stats.o info.o page.o rdma_transport.o ib_ring.o ib_rdma.o \
614                 ib_recv.o ib.o ib_send.o ib_sysctl.o ib_stats.o ib_cm.o \
615                 rdsv3_sc.o rdsv3_debug.o rdsv3_impl.o rdma.o rdsv3_af_thr.o

617 ISER_OBJS +=    iser.o iser_cm.o iser_cq.o iser_ib.o iser_idm.o \
618                 iser_resource.o iser_xfer.o

620 UDP_OBJS +=     udpddi.o

622 UDP6_OBJS +=    udp6ddi.o

624 DCCP_OBJS +=    dccpddi.o

626 DCCP6_OBJS +=   dccp6ddi.o

628 #endif /* ! codereview */
629 SY_OBJS +=      gentty.o

631 TCO_OBJS +=     ticots.o

633 TCOO_OBJS +=    ticotsord.o

635 TCL_OBJS +=     ticlts.o

637 TL_OBJS +=      tl.o

639 DUMP_OBJS +=    dump.o

641 BPF_OBJS +=     bpf.o bpf_filter.o bpf_mod.o bpf_dlt.o bpf_mac.o

643 CLONE_OBJS +=   clone.o

645 CN_OBJS +=      cons.o

647 DLD_OBJS +=     dld_drv.o dld_proto.o dld_str.o dld_flow.o

649 DLS_OBJS +=     dls.o dls_link.o dls_mod.o dls_stat.o dls_mgmt.o

651 GLD_OBJS +=     gld.o gldutil.o

653 MAC_OBJS +=     mac.o  mac_bcast.o mac_client.o mac_datapath_setup.o mac_flow.o
654                 mac_hio.o mac_mod.o mac_ndd.o mac_provider.o mac_sched.o \
```

```
655                 mac_protect.o mac_soft_ring.o mac_stat.o mac_util.o

657 MAC_6TO4_OBJS +=        mac_6to4.o

659 MAC_ETHER_OBJS +=       mac_ether.o

661 MAC_IPV4_OBJS +=        mac_ipv4.o

663 MAC_IPV6_OBJS +=        mac_ipv6.o

665 MAC_WIFI_OBJS +=        mac_wifi.o

667 MAC_IB_OBJS +=          mac_ib.o

669 IPTUN_OBJS +=   iptun_dev.o iptun_ctl.o iptun.o

671 AGGR_OBJS +=    aggr_dev.o aggr_ctl.o aggr_grp.o aggr_port.o \
672                 aggr_send.o aggr_recv.o aggr_lacp.o

674 SOFTMAC_OBJS += softmac_main.o softmac_ctl.o softmac_capab.o \
675                 softmac_dev.o softmac_stat.o softmac_pkt.o softmac_fp.o

677 NET80211_OBJS += net80211.o net80211_proto.o net80211_input.o \
678                  net80211_output.o net80211_node.o net80211_crypto.o \
679                  net80211_crypto_none.o net80211_crypto_wep.o net80211_ioctl.o \
680                  net80211_crypto_tkip.o net80211_crypto_ccmp.o  \
681                  net80211_ht.o

683 VNIC_OBJS +=    vnic_ctl.o vnic_dev.o

685 SIMNET_OBJS +=  simnet.o

687 IB_OBJS +=      ibnex.o ibnex_ioctl.o ibnex_hca.o

689 IBCM_OBJS +=    ibcm_impl.o ibcm_sm.o ibcm_ti.o ibcm_utils.o ibcm_path.o \
690                 ibcm_arp.o ibcm_arp_link.o

692 IBDM_OBJS +=    ibdm.o

694 IBDMA_OBJS +=   ibdma.o

696 IBMF_OBJS +=    ibmf.o ibmf_impl.o ibmf_dr.o ibmf_wqe.o ibmf_ud_dest.o ibmf_mod.
697                 ibmf_send.o ibmf_recv.o ibmf_handlers.o ibmf_trans.o \
698                 ibmf_timers.o ibmf_msg.o ibmf_utils.o ibmf_rmpp.o \
699                 ibmf_saa.o ibmf_saa_impl.o ibmf_saa_utils.o ibmf_saa_events.o

701 IBTL_OBJS +=    ibtl_impl.o ibtl_util.o ibtl_mem.o ibtl_handlers.o ibtl_qp.o \
702                 ibtl_cq.o ibtl_wr.o ibtl_hca.o ibtl_chan.o ibtl_cm.o \
703                 ibtl_mcg.o ibtl_ibnex.o ibtl_srq.o ibtl_part.o

705 TAVOR_OBJS +=   tavor.o tavor_agents.o tavor_cfg.o tavor_ci.o tavor_cmd.o \
706                 tavor_cq.o tavor_event.o tavor_ioctl.o tavor_misc.o \
707                 tavor_mr.o tavor_qp.o tavor_qpmod.o tavor_rsrc.o \
708                 tavor_srq.o tavor_stats.o tavor_umap.o tavor_wr.o

710 HERMON_OBJS +=  hermon.o hermon_agents.o hermon_cfg.o hermon_ci.o hermon_cmd.o \
711                 hermon_cq.o hermon_event.o hermon_ioctl.o hermon_misc.o \
712                 hermon_mr.o hermon_qp.o hermon_qpmod.o hermon_rsrc.o \
713                 hermon_srq.o hermon_stats.o hermon_umap.o hermon_wr.o \
714                 hermon_fcoib.o hermon_fm.o

716 DAPLT_OBJS +=   daplt.o

718 SOL_OFS_OBJS += sol_cma.o sol_ib_cma.o sol_uobj.o \
719                 sol_ofs_debug_util.o sol_ofs_gen_util.o \
720                 sol_kverbs.o
```



```
722 SOL_UCMA_OBJS +=        sol_ucma.o

724 SOL_UVERBS_OBJS +=      sol_uverbs.o sol_uverbs_comp.o sol_uverbs_event.o \
725                         sol_uverbs_hca.o sol_uverbs_qp.o

727 SOL_UMAD_OBJS += sol_umad.o

729 KSTAT_OBJS +=   kstat.o

731 KSYMS_OBJS +=   ksyms.o

733 INSTANCE_OBJS += inst_sync.o

735 IWSCN_OBJS +=   iwscons.o

737 LOFI_OBJS +=    lofi.o LzmaDec.o

739 FSSNAP_OBJS +=  fssnap.o

741 FSSNAPIF_OBJS += fssnap_if.o

743 MM_OBJS +=      mem.o

745 PHYSMEM_OBJS += physmem.o

747 OPTIONS_OBJS += options.o

749 WINLOCK_OBJS += winlockio.o

751 PM_OBJS +=      pm.o
752 SRN_OBJS +=     srn.o

754 PSEUDO_OBJS +=  pseudonex.o

756 RAMDISK_OBJS += ramdisk.o

758 LLC1_OBJS += llc1.o

760 USBKBM_OBJS += usbkbm.o

762 USBWCM_OBJS += usbwcm.o

764 BOFI_OBJS += bofi.o

766 HID_OBJS += hid.o

768 HWA_RC_OBJS += hwarc.o

770 USBSKEL_OBJS += usbskel.o

772 USBVC_OBJS += usbvc.o usbvc_v4l2.o

774 HIDPARSER_OBJS += hidparser.o

776 USB_AC_OBJS += usb_ac.o

778 USB_AS_OBJS += usb_as.o

780 USB_AH_OBJS += usb_ah.o

782 USBMS_OBJS += usbms.o

784 USBPRN_OBJS += usbprn.o

786 UGEN_OBJS += ugen.o
```

```
788 USBSER_OBJS += usbser.o usbser_rseq.o

790 USBSACM_OBJS += usbsacm.o

792 USBSER_KEYSPAN_OBJS += usbser_keyspan.o keyspan_dsd.o keyspan_pipe.o

794 USBS49_FW_OBJS += keyspan_49fw.o

796 USBSPRL_OBJS += usbser_pl2303.o pl2303_dsd.o

798 WUSB_CA_OBJS += wusb_ca.o

800 USBFTDI_OBJS += usbser_uftdi.o uftdi_dsd.o

802 USBECM_OBJS += usbecm.o

804 WC_OBJS += wscons.o vcons.o

806 VCONS_CONF_OBJS += vcons_conf.o

808 SCSI_OBJS +=    scsi_capabilities.o scsi_confsubr.o scsi_control.o \
809                 scsi_data.o scsi_fm.o scsi_hba.o scsi_reset_notify.o \
810                 scsi_resource.o scsi_subr.o scsi_transport.o scsi_watch.o \
811                 smp_transport.o

813 SCSI_VHCI_OBJS +=               scsi_vhci.o mpapi_impl.o scsi_vhci_tpgs.o

815 SCSI_VHCI_F_SYM_OBJS +=         sym.o

817 SCSI_VHCI_F_TPGS_OBJS +=        tpgs.o

819 SCSI_VHCI_F_ASYM_SUN_OBJS +=    asym_sun.o

821 SCSI_VHCI_F_SYM_HDS_OBJS +=     sym_hds.o

823 SCSI_VHCI_F_TAPE_OBJS +=        tape.o

825 SCSI_VHCI_F_TPGS_TAPE_OBJS +=   tpgs_tape.o

827 SGEN_OBJS +=    sgen.o

829 SMP_OBJS +=     smp.o

831 SATA_OBJS +=    sata.o

833 USBA_OBJS +=    hcdi.o  usba.o  usbai.o hubdi.o parser.o genconsole.o \
834                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
835                 usba_devdb.o usba10_calls.o usba_ugen.o whcdi.o wa.o
836 USBA_WITHOUT_WUSB_OBJS +=       hcdi.o  usba.o  usbai.o hubdi.o parser.o gencons
837                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
838                 usba_devdb.o usba10_calls.o usba_ugen.o

840 USBA10_OBJS +=  usba10.o

842 RSM_OBJS +=     rsm.o   rsmka_pathmanager.o     rsmka_util.o

844 RSMOPS_OBJS +=  rsmops.o

846 S1394_OBJS +=   t1394.o t1394_errmsg.o s1394.o s1394_addr.o s1394_asynch.o \
847                 s1394_bus_reset.o s1394_cmp.o s1394_csr.o s1394_dev_disc.o \
848                 s1394_fa.o s1394_fcp.o \
849                 s1394_hotplug.o s1394_isoch.o s1394_misc.o h1394.o nx1394.o

851 HCI1394_OBJS += hci1394.o hci1394_async.o hci1394_attach.o hci1394_buf.o \
852                 hci1394_csr.o hci1394_detach.o hci1394_extern.o \
```



```
853                 hci1394_ioctl.o hci1394_isoch.o hci1394_isr.o \
854                 hci1394_ixl_comp.o hci1394_ixl_isr.o hci1394_ixl_misc.o \
855                 hci1394_ixl_update.o hci1394_misc.o hci1394_ohci.o \
856                 hci1394_q.o hci1394_s1394if.o hci1394_tlabel.o \
857                 hci1394_tlist.o hci1394_vendor.o

859 AV1394_OBJS +=  av1394.o av1394_as.o av1394_async.o av1394_cfgrom.o \
860                 av1394_cmp.o av1394_fcp.o av1394_isoch.o av1394_isoch_chan.o \
861                 av1394_isoch_recv.o av1394_isoch_xmit.o av1394_list.o \
862                 av1394_queue.o

864 DCAM1394_OBJS += dcam.o dcam_frame.o dcam_param.o dcam_reg.o \
865                 dcam_ring_buff.o

867 SCSA1394_OBJS += hba.o sbp2_driver.o sbp2_bus.o

869 SBP2_OBJS += cfgrom.o sbp2.o

871 PMODEM_OBJS += pmodem.o pmodem_cis.o cis.o cis_callout.o cis_handlers.o cis_para

873 DSW_OBJS +=     dsw.o dsw_dev.o ii_tree.o

875 NCALL_OBJS +=   ncall.o \
876                 ncall_stub.o

878 RDC_OBJS +=     rdc.o \
879                 rdc_dev.o \
880                 rdc_io.o \
881                 rdc_clnt.o \
882                 rdc_prot_xdr.o \
883                 rdc_svc.o \
884                 rdc_bitmap.o \
885                 rdc_health.o \
886                 rdc_subr.o \
887                 rdc_diskq.o

889 RDCSRV_OBJS +=  rdcsrv.o

891 RDCSTUB_OBJS += rdc_stub.o

893 SDBC_OBJS +=    sd_bcache.o \
894                 sd_bio.o \
895                 sd_conf.o \
896                 sd_ft.o \
897                 sd_hash.o \
898                 sd_io.o \
899                 sd_misc.o \
900                 sd_pcu.o \
901                 sd_tdaemon.o \
902                 sd_trace.o \
903                 sd_iob_impl0.o \
904                 sd_iob_impl1.o \
905                 sd_iob_impl2.o \
906                 sd_iob_impl3.o \
907                 sd_iob_impl4.o \
908                 sd_iob_impl5.o \
909                 sd_iob_impl6.o \
910                 sd_iob_impl7.o \
911                 safestore.o \
912                 safestore_ram.o

914 NSCTL_OBJS +=   nsctl.o \
915                 nsc_cache.o \
916                 nsc_disk.o \
917                 nsc_dev.o \
918                 nsc_freeze.o \
```

```
                nsc_gen.o \
                nsc_mem.o \
                nsc_ncallio.o \
                nsc_power.o \
                nsc_resv.o \
                nsc_rmspin.o \
                nsc_solaris.o \
                nsc_trap.o \
                nsc_list.o
UNISTAT_OBJS += spuni.o \
                spcs_s_k.o

NSKERN_OBJS +=  nsc_ddi.o \
                nsc_proc.o \
                nsc_raw.o \
                nsc_thread.o \
                nskernd.o

SV_OBJS +=      sv.o

PMCS_OBJS += pmcs_attach.o pmcs_ds.o pmcs_intr.o pmcs_nvram.o pmcs_sata.o \
                pmcs_scsa.o pmcs_smhba.o pmcs_subr.o pmcs_fwlog.o

PMCS8001FW_C_OBJS +=    pmcs_fw_hdr.o
PMCS8001FW_OBJS +=              $(PMCS8001FW_C_OBJS) SPCBoot.o ila.o firmware.o

#
#       Build up defines and paths.

ST_OBJS +=      st.o    st_conf.o

EMLXS_OBJS +=   emlxs_clock.o emlxs_dfc.o emlxs_dhchap.o emlxs_diag.o \
                emlxs_download.o emlxs_dump.o emlxs_els.o emlxs_event.o \
                emlxs_fcf.o emlxs_fcp.o emlxs_fct.o emlxs_hba.o emlxs_ip.o \
                emlxs_mbox.o emlxs_mem.o emlxs_msg.o emlxs_node.o \
                emlxs_pkt.o emlxs_sli3.o emlxs_sli4.o emlxs_solaris.o \
                emlxs_thread.o

EMLXS_FW_OBJS +=        emlxs_fw.o

OCE_OBJS +=     oce_buf.o oce_fm.o oce_gld.o oce_hw.o oce_intr.o oce_main.o \
                oce_mbx.o oce_mq.o oce_queue.o oce_rx.o oce_stat.o oce_tx.o \
                oce_utils.o

FCT_OBJS += discovery.o fct.o

QLT_OBJS += 2400.o 2500.o 8100.o qlt.o qlt_dma.o

SRPT_OBJS += srpt_mod.o srpt_ch.o srpt_cm.o srpt_ioc.o srpt_stp.o

FCOE_OBJS += fcoe.o fcoe_eth.o fcoe_fc.o

FCOET_OBJS += fcoet.o fcoet_eth.o fcoet_fc.o

FCOEI_OBJS += fcoei.o fcoei_eth.o fcoei_lv.o

ISCSIT_SHARED_OBJS += \
                iscsit_common.o

ISCSIT_OBJS +=  $(ISCSIT_SHARED_OBJS) \
                iscsit.o iscsit_tgt.o iscsit_sess.o iscsit_login.o \
                iscsit_text.o iscsit_isns.o iscsit_radiusauth.o \
                iscsit_radiuspacket.o iscsit_auth.o iscsit_authclient.o

PPPT_OBJS +=    alua_ic_if.o pppt.o pppt_msg.o pppt_tgt.o
```



```
STMF_OBJS += lun_map.o stmf.o

STMF_SBD_OBJS += sbd.o sbd_scsi.o sbd_pgr.o sbd_zvol.o

SYSMSG_OBJS +=  sysmsg.o

SES_OBJS +=     ses.o ses_sen.o ses_safte.o ses_ses.o

TNF_OBJS +=     tnf_buf.o       tnf_trace.o     tnf_writer.o    trace_init.o \
                trace_funcs.o   tnf_probe.o     tnf.o

LOGINDMUX_OBJS += logindmux.o

DEVINFO_OBJS += devinfo.o

DEVPOLL_OBJS += devpoll.o

DEVPOOL_OBJS += devpool.o

I8042_OBJS +=   i8042.o

KB8042_OBJS +=  \
                at_keyprocess.o \
                kb8042.o        \
                kb8042_keytables.o

MOUSE8042_OBJS += mouse8042.o

FDC_OBJS +=     fdc.o

ASY_OBJS +=     asy.o

ECPP_OBJS +=    ecpp.o

VUIDM3P_OBJS += vuidmice.o vuidm3p.o

VUIDM4P_OBJS += vuidmice.o vuidm4p.o

VUIDM5P_OBJS += vuidmice.o vuidm5p.o

VUIDPS2_OBJS += vuidmice.o vuidps2.o

HPCSVC_OBJS += hpcsvc.o

PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehpc.o pcishpc.o pcie_pwr.o p

PCIHPNEXUS_OBJS += pcihp.o

OPENEEPR_OBJS += openprom.o

RANDOM_OBJS += random.o

PSHOT_OBJS += pshot.o

GEN_DRV_OBJS += gen_drv.o

TCLIENT_OBJS += tclient.o

TPHCI_OBJS += tphci.o

TVHCI_OBJS += tvhci.o

EMUL64_OBJS += emul64.o emul64_bsd.o

FCP_OBJS += fcp.o
```

```
1051 FCIP_OBJS += fcip.o

1053 FCSM_OBJS += fcsm.o

1055 FCTL_OBJS += fctl.o

1057 FP_OBJS += fp.o

1059 QLC_OBJS += ql_api.o ql_debug.o ql_hba_fru.o ql_init.o ql_iocb.o ql_ioctl.o \
1060         ql_isr.o ql_mbx.o ql_nx.o ql_xioctl.o ql_fw_table.o

1062 QLC_FW_2200_OBJS += ql_fw_2200.o

1064 QLC_FW_2300_OBJS += ql_fw_2300.o

1066 QLC_FW_2400_OBJS += ql_fw_2400.o

1068 QLC_FW_2500_OBJS += ql_fw_2500.o

1070 QLC_FW_6322_OBJS += ql_fw_6322.o

1072 QLC_FW_8100_OBJS += ql_fw_8100.o

1074 QLGE_OBJS += qlge.o qlge_dbg.o qlge_flash.o qlge_fm.o qlge_gld.o qlge_mpi.o

1076 ZCONS_OBJS += zcons.o

1078 NV_SATA_OBJS += nv_sata.o

1080 SI3124_OBJS += si3124.o

1082 AHCI_OBJS += ahci.o

1084 PCIIDE_OBJS += pci-ide.o

1086 PCEPP_OBJS += pcepp.o

1088 CPC_OBJS += cpc.o

1090 CPUID_OBJS += cpuid_drv.o

1092 SYSEVENT_OBJS += sysevent.o

1094 BL_OBJS += bl.o

1096 DRM_OBJS += drm_sunmod.o drm_kstat.o drm_agpsupport.o \
1097             drm_auth.o drm_bufs.o drm_context.o drm_dma.o \
1098             drm_drawable.o drm_drv.o drm_fops.o drm_ioctl.o drm_irq.o \
1099             drm_lock.o drm_memory.o drm_msg.o drm_pci.o drm_scatter.o \
1100             drm_cache.o drm_gem.o drm_mm.o ati_pcigart.o

1102 FM_OBJS += devfm.o devfm_machdep.o

1104 RTLS_OBJS +=    rtls.o

1106 #
1107 #                       exec modules
1108 #
1109 AOUTEXEC_OBJS +=aout.o

1111 ELFEXEC_OBJS += elf.o elf_notes.o old_notes.o

1113 INTPEXEC_OBJS +=intp.o

1115 SHBINEXEC_OBJS +=shbin.o
```



```
1117 JAVAEXEC_OBJS +=java.o

1119 #
1120 #                       file system modules
1121 #
1122 AUTOFS_OBJS +=  auto_vfsops.o auto_vnops.o auto_subr.o auto_xdr.o auto_sys.o

1124 CACHEFS_OBJS += cachefs_cnode.o         cachefs_cod.o \
1125                 cachefs_dir.o           cachefs_dlog.o  cachefs_filegrp.o \
1126                 cachefs_fscache.o       cachefs_ioctl.o cachefs_log.o \
1127                 cachefs_module.o \
1128                 cachefs_noopc.o         cachefs_resource.o \
1129                 cachefs_strict.o \
1130                 cachefs_subr.o          cachefs_vfsops.o \
1131                 cachefs_vnops.o

1133 DCFS_OBJS +=    dc_vnops.o

1135 DEVFS_OBJS +=   devfs_subr.o    devfs_vfsops.o  devfs_vnops.o

1137 DEV_OBJS  +=    sdev_subr.o     sdev_vfsops.o   sdev_vnops.o    \
1138                 sdev_ptsops.o   sdev_zvolops.o  sdev_comm.o     \
1139                 sdev_profile.o  sdev_ncache.o   sdev_netops.o   \
1140                 sdev_ipnetops.o \
1141                 sdev_vtops.o

1143 CTFS_OBJS +=    ctfs_all.o ctfs_cdir.o ctfs_ctl.o ctfs_event.o \
1144                 ctfs_latest.o ctfs_root.o ctfs_sym.o ctfs_tdir.o ctfs_tmpl.o

1146 OBJFS_OBJS +=   objfs_vfs.o     objfs_root.o    objfs_common.o \
1147                 objfs_odir.o    objfs_data.o

1149 FDFS_OBJS +=    fdops.o

1151 FIFO_OBJS +=    fifosubr.o      fifovnops.o

1153 PIPE_OBJS +=    pipe.o

1155 HSFS_OBJS +=    hsfs_node.o     hsfs_subr.o     hsfs_vfsops.o   hsfs_vnops.o \
1156                 hsfs_susp.o     hsfs_rrip.o     hsfs_susp_subr.o

1158 LOFS_OBJS +=    lofs_subr.o     lofs_vfsops.o   lofs_vnops.o

1160 NAMEFS_OBJS +=  namevfs.o       namevno.o

1162 NFS_OBJS +=     nfs_client.o    nfs_common.o    nfs_dump.o \
1163                 nfs_subr.o      nfs_vfsops.o    nfs_vnops.o \
1164                 nfs_xdr.o       nfs_sys.o       nfs_strerror.o \
1165                 nfs3_vfsops.o   nfs3_vnops.o    nfs3_xdr.o \
1166                 nfs_acl_vnops.o nfs_acl_xdr.o   nfs4_vfsops.o \
1167                 nfs4_vnops.o    nfs4_xdr.o      nfs4_idmap.o \
1168                 nfs4_shadow.o   nfs4_subr.o \
1169                 nfs4_attr.o     nfs4_rnode.o    nfs4_client.o \
1170                 nfs4_acache.o   nfs4_common.o   nfs4_client_state.o \
1171                 nfs4_callback.o nfs4_recovery.o nfs4_client_secinfo.o \
1172                 nfs4_client_debug.o     nfs_stats.o \
1173                 nfs4_acl.o      nfs4_stub_vnops.o       nfs_cmd.o

1175 NFSSRV_OBJS +=  nfs_server.o    nfs_srv.o       nfs3_srv.o \
1176                 nfs_acl_srv.o   nfs_auth.o      nfs_auth_xdr.o \
1177                 nfs_export.o    nfs_log.o       nfs_log_xdr.o \
1178                 nfs4_srv.o      nfs4_state.o    nfs4_srv_attr.o \
1179                 nfs4_srv_ns.o   nfs4_db.o       nfs4_srv_deleg.o \
1180                 nfs4_deleg_ops.o nfs4_srv_readdir.o nfs4_dispatch.o

1182 SMBSRV_SHARED_OBJS += \
```

```
1183                smb_inet.o \
1184                smb_match.o \
1185                smb_msgbuf.o \
1186                smb_oem.o \
1187                smb_string.o \
1188                smb_utf8.o \
1189                smb_door_legacy.o \
1190                smb_xdr.o \
1191                smb_token.o \
1192                smb_token_xdr.o \
1193                smb_sid.o \
1194                smb_native.o \
1195                smb_netbios_util.o

1197 SMBSRV_OBJS +=  $(SMBSRV_SHARED_OBJS)                  \
1198                smb_acl.o                               \
1199                smb_alloc.o                             \
1200                smb_close.o                             \
1201                smb_common_open.o                       \
1202                smb_common_transact.o                   \
1203                smb_create.o                            \
1204                smb_delete.o                            \
1205                smb_directory.o                         \
1206                smb_dispatch.o                          \
1207                smb_echo.o                              \
1208                smb_fem.o                               \
1209                smb_find.o                              \
1210                smb_flush.o                             \
1211                smb_fsinfo.o                            \
1212                smb_fsops.o                             \
1213                smb_init.o                              \
1214                smb_kdoor.o                             \
1215                smb_kshare.o                            \
1216                smb_kutil.o                             \
1217                smb_lock.o                              \
1218                smb_lock_byte_range.o                   \
1219                smb_locking_andx.o                      \
1220                smb_logoff_andx.o                       \
1221                smb_mangle_name.o                       \
1222                smb_mbuf_marshaling.o                   \
1223                smb_mbuf_util.o                         \
1224                smb_negotiate.o                         \
1225                smb_net.o                               \
1226                smb_node.o                              \
1227                smb_nt_cancel.o                         \
1228                smb_nt_create_andx.o                    \
1229                smb_nt_transact_create.o                \
1230                smb_nt_transact_ioctl.o                 \
1231                smb_nt_transact_notify_change.o         \
1232                smb_nt_transact_quota.o                 \
1233                smb_nt_transact_security.o              \
1234                smb_odir.o                              \
1235                smb_ofile.o                             \
1236                smb_open_andx.o                         \
1237                smb_opipe.o                             \
1238                smb_oplock.o                            \
1239                smb_pathname.o                          \
1240                smb_print.o                             \
1241                smb_process_exit.o                      \
1242                smb_query_fileinfo.o                    \
1243                smb_read.o                              \
1244                smb_rename.o                            \
1245                smb_sd.o                                \
1246                smb_seek.o                              \
1247                smb_server.o                            \
1248                smb_session.o                           \
```



```
1249                smb_session_setup_andx.o                \
1250                smb_set_fileinfo.o                      \
1251                smb_signing.o                           \
1252                smb_tree.o                              \
1253                smb_trans2_create_directory.o           \
1254                smb_trans2_dfs.o                        \
1255                smb_trans2_find.o                       \
1256                smb_tree_connect.o                      \
1257                smb_unlock_byte_range.o                 \
1258                smb_user.o                              \
1259                smb_vfs.o                               \
1260                smb_vops.o                              \
1261                smb_vss.o                               \
1262                smb_write.o                             \
1263                smb_write_raw.o

1265 PCFS_OBJS +=    pc_alloc.o      pc_dir.o        pc_node.o       pc_subr.o \
1266                 pc_vfsops.o     pc_vnops.o

1268 PROC_OBJS +=    prcontrol.o     prioctl.o       prsubr.o        prusrio.o \
1269                 prvfsops.o      prvnops.o

1271 MNTFS_OBJS +=   mntvfsops.o     mntvnops.o

1273 SHAREFS_OBJS += sharetab.o      sharefs_vfsops.o        sharefs_vnops.o

1275 SPEC_OBJS +=    specsubr.o      specvfsops.o    specvnops.o

1277 SOCK_OBJS +=    socksubr.o      sockvfsops.o    sockparams.o    \
1278                 socksyscalls.o  socktpi.o       sockstr.o \
1279                 sockcommon_vnops.o      sockcommon_subr.o \
1280                 sockcommon_sops.o       sockcommon.o    \
1281                 sock_notsupp.o  socknotify.o \
1282                 nl7c.o          nl7curi.o       nl7chttp.o      nl7clogd.o \
1283                 nl7cnca.o       sodirect.o      sockfilter.o

1285 TMPFS_OBJS +=   tmp_dir.o       tmp_subr.o      tmp_tnode.o     tmp_vfsops.o \
1286                 tmp_vnops.o

1288 UDFS_OBJS +=    udf_alloc.o     udf_bmap.o      udf_dir.o       \
1289                 udf_inode.o     udf_subr.o      udf_vfsops.o    \
1290                 udf_vnops.o

1292 UFS_OBJS +=     ufs_alloc.o     ufs_bmap.o      ufs_dir.o       ufs_xattr.o \
1293                 ufs_inode.o     ufs_subr.o      ufs_tables.o    ufs_vfsops.o \
1294                 ufs_vnops.o     quota.o         quotacalls.o    quota_ufs.o \
1295                 ufs_filio.o     ufs_lockfs.o    ufs_thread.o    ufs_trans.o \
1296                 ufs_acl.o       ufs_panic.o     ufs_directio.o  ufs_log.o \
1297                 ufs_extvnops.o  ufs_snap.o      lufs.o          lufs_thread.o \
1298                 lufs_log.o      lufs_map.o      lufs_top.o      lufs_debug.o
1299 VSCAN_OBJS +=   vscan_drv.o     vscan_svc.o vscan_door.o

1301 NSMB_OBJS +=    smb_conn.o      smb_dev.o       smb_iod.o       smb_pass.o \
1302                 smb_rq.o        smb_sign.o      smb_smb.o       smb_subrs.o \
1303                 smb_time.o      smb_tran.o      smb_trantcp.o   smb_usr.o \
1304                 subr_mchain.o

1306 SMBFS_COMMON_OBJS += smbfs_ntacl.o
1307 SMBFS_OBJS +=   smbfs_vfsops.o  smbfs_vnops.o   smbfs_node.o    \
1308                 smbfs_acl.o     smbfs_client.o  smbfs_smb.o     \
1309                 smbfs_subr.o    smbfs_subr2.o   \
1310                 smbfs_rwlock.o  smbfs_xattr.o   \
1311                 $(SMBFS_COMMON_OBJS)


1314 #
```

```
1315 #                       LVM modules
1316 #
1317 MD_OBJS += md.o md_error.o md_ioctl.o md_mddb.o md_names.o \
1318         md_med.o md_rename.o md_subr.o

1320 MD_COMMON_OBJS = md_convert.o md_crc.o md_revchk.o

1322 MD_DERIVED_OBJS = metamed_xdr.o meta_basic_xdr.o

1324 SOFTPART_OBJS += sp.o sp_ioctl.o

1326 STRIPE_OBJS += stripe.o stripe_ioctl.o

1328 HOTSPARES_OBJS += hotspares.o

1330 RAID_OBJS += raid.o raid_ioctl.o raid_replay.o raid_resync.o raid_hotspare.o

1332 MIRROR_OBJS += mirror.o mirror_ioctl.o mirror_resync.o

1334 NOTIFY_OBJS += md_notify.o

1336 TRANS_OBJS += mdtrans.o trans_ioctl.o trans_log.o

1338 ZFS_COMMON_OBJS +=              \
1339         arc.o                   \
1340         bplist.o                \
1341         bpobj.o                 \
1342         bptree.o                \
1343         dbuf.o                  \
1344         ddt.o                   \
1345         ddt_zap.o               \
1346         dmu.o                   \
1347         dmu_diff.o              \
1348         dmu_send.o              \
1349         dmu_object.o            \
1350         dmu_objset.o            \
1351         dmu_traverse.o          \
1352         dmu_tx.o                \
1353         dnode.o                 \
1354         dnode_sync.o            \
1355         dsl_dir.o               \
1356         dsl_dataset.o           \
1357         dsl_deadlist.o          \
1358         dsl_pool.o              \
1359         dsl_synctask.o          \
1360         dmu_zfetch.o            \
1361         dsl_deleg.o             \
1362         dsl_prop.o              \
1363         dsl_scan.o              \
1364         zfeature.o              \
1365         gzip.o                  \
1366         lzjb.o                  \
1367         metaslab.o              \
1368         refcount.o              \
1369         sa.o                    \
1370         sha256.o                \
1371         spa.o                   \
1372         spa_config.o            \
1373         spa_errlog.o            \
1374         spa_history.o           \
1375         spa_misc.o              \
1376         space_map.o             \
1377         txg.o                   \
1378         uberblock.o             \
1379         unique.o                \
1380         vdev.o                  \
```



```
1381         vdev_cache.o            \
1382         vdev_file.o             \
1383         vdev_label.o            \
1384         vdev_mirror.o           \
1385         vdev_missing.o          \
1386         vdev_queue.o            \
1387         vdev_raidz.o            \
1388         vdev_root.o             \
1389         zap.o                   \
1390         zap_leaf.o              \
1391         zap_micro.o             \
1392         zfs_byteswap.o          \
1393         zfs_debug.o             \
1394         zfs_fm.o                \
1395         zfs_fuid.o              \
1396         zfs_sa.o                \
1397         zfs_znode.o             \
1398         zil.o                   \
1399         zio.o                   \
1400         zio_checksum.o          \
1401         zio_compress.o          \
1402         zio_inject.o            \
1403         zle.o                   \
1404         zrlock.o

1406 ZFS_SHARED_OBJS +=              \
1407         zfeature_common.o       \
1408         zfs_comutil.o           \
1409         zfs_deleg.o             \
1410         zfs_fletcher.o          \
1411         zfs_namecheck.o         \
1412         zfs_prop.o              \
1413         zpool_prop.o            \
1414         zprop_common.o

1416 ZFS_OBJS +=                     \
1417         $(ZFS_COMMON_OBJS)      \
1418         $(ZFS_SHARED_OBJS)      \
1419         vdev_disk.o             \
1420         zfs_acl.o               \
1421         zfs_ctldir.o            \
1422         zfs_dir.o               \
1423         zfs_ioctl.o             \
1424         zfs_log.o               \
1425         zfs_onexit.o            \
1426         zfs_replay.o            \
1427         zfs_rlock.o             \
1428         rrwlock.o               \
1429         zfs_vfsops.o            \
1430         zfs_vnops.o             \
1431         zvol.o

1433 ZUT_OBJS +=                     \
1434         zut.o

1436 #
1437 #                       streams modules
1438 #
1439 BUFMOD_OBJS     +=      bufmod.o

1441 CONNLD_OBJS +=  connld.o

1443 DEDUMP_OBJS +=  dedump.o

1445 DRCOMPAT_OBJS +=        drcompat.o
```

```
1447 LDLINUX_OBJS += ldlinux.o

1449 LDTERM_OBJS +=  ldterm.o uwidth.o

1451 PCKT_OBJS +=    pckt.o

1453 PFMOD_OBJS +=   pfmod.o

1455 PTEM_OBJS +=    ptem.o

1457 REDIRMOD_OBJS += strredirm.o

1459 TIMOD_OBJS +=   timod.o

1461 TIRDWR_OBJS +=  tirdwr.o

1463 TTCOMPAT_OBJS +=ttcompat.o

1465 LOG_OBJS +=     log.o

1467 PIPEMOD_OBJS += pipemod.o

1469 RPCMOD_OBJS +=  rpcmod.o        clnt_cots.o     clnt_clts.o \
1470                 clnt_gen.o      clnt_perr.o     mt_rpcinit.o    rpc_calmsg.o \
1471                 rpc_prot.o      rpc_sztypes.o   rpc_subr.o      rpcb_prot.o \
1472                 svc.o           svc_clts.o      svc_gen.o       svc_cots.o \
1473                 rpcsys.o        xdr_sizeof.o    clnt_rdma.o     svc_rdma.o \
1474                 xdr_rdma.o      rdma_subr.o     xdrrdma_sizeof.o

1476 TLIMOD_OBJS +=  tlimod.o        t_kalloc.o      t_kbind.o       t_kclose.o \
1477                 t_kconnect.o    t_kfree.o       t_kgtstate.o    t_kopen.o \
1478                 t_krcvudat.o    t_ksndudat.o    t_kspoll.o      t_kunbind.o \
1479                 t_kutil.o

1481 RLMOD_OBJS += rlmod.o

1483 TELMOD_OBJS += telmod.o

1485 CRYPTMOD_OBJS += cryptmod.o

1487 KB_OBJS +=      kbd.o           keytables.o

1489 #
1490 #                       ID mapping module
1491 #
1492 IDMAP_OBJS +=   idmap_mod.o     idmap_kapi.o    idmap_xdr.o     idmap_cache.o

1494 #
1495 #                       scheduling class modules
1496 #
1497 SDC_OBJS +=             sysdc.o

1499 RT_OBJS +=              rt.o
1500 RT_DPTBL_OBJS +=        rt_dptbl.o

1502 TS_OBJS +=              ts.o
1503 TS_DPTBL_OBJS +=        ts_dptbl.o

1505 IA_OBJS +=              ia.o

1507 FSS_OBJS +=             fss.o

1509 FX_OBJS +=              fx.o
1510 FX_DPTBL_OBJS +=        fx_dptbl.o

1512 #
```



```
1513 #                       Inter-Process Communication (IPC) modules
1514 #
1515 IPC_OBJS +=     ipc.o

1517 IPCMSG_OBJS +=  msg.o

1519 IPCSEM_OBJS +=  sem.o

1521 IPCSHM_OBJS +=  shm.o

1523 #
1524 #                       bignum module
1525 #
1526 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o

1528 BIGNUM_OBJS += $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)

1530 #
1531 #                       kernel cryptographic framework
1532 #
1533 KCF_OBJS +=     kcf.o kcf_callprov.o kcf_cbufcall.o kcf_cipher.o kcf_crypto.o \
1534                 kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1535                 kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1536                 kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1537                 kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1538                 kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1539                 fips_random.o

1541 CRYPTOADM_OBJS += cryptoadm.o

1543 CRYPTO_OBJS +=  crypto.o

1545 DPROV_OBJS +=   dprov.o

1547 DCA_OBJS +=     dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1548                 dca_rsa.o

1550 AESPROV_OBJS += aes.o aes_impl.o aes_modes.o

1552 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o

1554 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o

1556 ECCPROV_OBJS += ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1557                 ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1558                 ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1559                 ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1560                 mpi.o mplogic.o mpmontg.o mpprime.o oid.o \
1561                 secitem.o ec2_test.o ecp_test.o

1563 RSAPROV_OBJS += rsa.o rsa_impl.o pkcs1.o

1565 SWRANDPROV_OBJS += swrand.o

1567 #
1568 #                       kernel SSL
1569 #
1570 KSSL_OBJS +=    kssl.o kssloctl.o

1572 KSSL_SOCKFIL_MOD_OBJS += ksslfilter.o ksslapi.o ksslrec.o

1574 #
1575 #                       misc. modules
1576 #

1578 C2AUDIT_OBJS += adr.o audit.o audit_event.o audit_io.o \
```

```
1579                audit_path.o audit_start.o audit_syscalls.o audit_token.o \
1580                audit_mem.o

1582 PCIC_OBJS +=    pcic.o

1584 RPCSEC_OBJS +=  secmod.o        sec_clnt.o      sec_svc.o       sec_gen.o \
1585                auth_des.o      auth_kern.o     auth_none.o     auth_loopb.o\
1586                authdesprt.o    authdesubr.o    authu_prot.o \
1587                key_call.o      key_prot.o      svc_authu.o     svcauthdes.o

1589 RPCSEC_GSS_OBJS +=      rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
1590                rpcsec_gss_utils.o svc_rpcsec_gss.o

1592 CONSCONFIG_OBJS += consconfig.o

1594 CONSCONFIG_DACF_OBJS  += consconfig_dacf.o consplat.o

1596 TEM_OBJS += tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1598 KBTRANS_OBJS +=                        \
1599                kbtrans.o               \
1600                kbtrans_keytables.o     \
1601                kbtrans_polled.o        \
1602                kbtrans_streams.o       \
1603                usb_keytables.o

1605 KGSSD_OBJS +=   gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1606                gss_display_name.o gss_release_name.o gss_import_name.o \
1607                gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1609 KGSSD_DERIVED_OBJS = gssd_xdr.o

1611 KGSS_DUMMY_OBJS += dmech.o

1613 KSOCKET_OBJS += ksocket.o ksocket_mod.o

1615 CRYPTO= cksumtypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1616         nfold.o verify_checksum.o prng.o block_size.o make_checksum.o\
1617         checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1619 # crypto/des
1620 CRYPTO_DES= f_cbc.o f_cksum.o f_parity.o weak_key.o d3_cbc.o ef_crypto.o

1622 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1624 CRYPTO_ARCFOUR= k5_arcfour.o

1626 # crypto/enc_provider
1627 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o

1629 # crypto/hash_provider
1630 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_ksha1.o

1632 # crypto/keyhash_provider
1633 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1635 # crypto/crc32
1636 CRYPTO_CRC32= crc32.o

1638 # crypto/old
1639 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1641 # crypto/raw
1642 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1644 K5_KRB= kfree.o copy_key.o \
```



```
1645         parse.o init_ctx.o \
1646         ser_adata.o ser_addr.o \
1647         ser_auth.o ser_cksum.o \
1648         ser_key.o ser_princ.o \
1649         serialize.o unparse.o \
1650         ser_actx.o

1652 K5_OS=  timeofday.o toffset.o \
1653         init_os_ctx.o c_ustime.o

1655 SEAL=
1656 # EXPORT DELETE START
1657 SEAL=   seal.o unseal.o
1658 # EXPORT DELETE END

1660 MECH=   delete_sec_context.o \
1661         import_sec_context.o \
1662         gssapi_krb5.o \
1663         k5seal.o k5unseal.o k5sealv3.o \
1664         ser_sctx.o \
1665         sign.o \
1666         util_crypt.o  \
1667         util_validate.o  util_ordering.o  \
1668         util_seqnum.o util_set.o util_seed.o \
1669         wrap_size_limit.o verify.o



1673 MECH_GEN= util_token.o


1676 KGSS_KRB5_OBJS += krb5mech.o \
1677         $(MECH) $(SEAL) $(MECH_GEN) \
1678         $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1679         $(CRYPTO_ENC) $(CRYPTO_HASH) \
1680         $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1681         $(CRYPTO_OLD) \
1682         $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1684 DES_OBJS +=     des_crypt.o des_impl.o des_ks.o des_soft.o

1686 DLBOOT_OBJS +=  bootparam_xdr.o nfs_dlinet.o scan.o

1688 KRTLD_OBJS +=   kobj_bootflags.o getoptstr.o \
1689                 kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1691 MOD_OBJS +=     modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1693 STRPLUMB_OBJS += strplumb.o

1695 CPR_OBJS +=     cpr_driver.o cpr_dump.o \
1696                 cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1697                 cpr_uthread.o

1699 PROF_OBJS +=    prf.o

1701 SE_OBJS += se_driver.o

1703 SYSACCT_OBJS += acct.o

1705 ACCTCTL_OBJS += acctctl.o

1707 EXACCTSYS_OBJS += exacctsys.o

1709 KAIO_OBJS += aio.o
```

```
1711 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o

1713 BUSRA_OBJS += busra.o

1715 PCS_OBJS += pcs.o

1717 PCAN_OBJS += pcan.o

1719 PCATA_OBJS += pcide.o pcdisk.o pclabel.o pcata.o

1721 PCSER_OBJS += pcser.o pcser_cis.o

1723 PCWL_OBJS += pcwl.o

1725 PSET_OBJS +=    pset.o

1727 OHCI_OBJS += ohci.o ohci_hub.o ohci_polled.o

1729 UHCI_OBJS += uhci.o uhciutil.o uhcitgt.o uhcihub.o uhcipolled.o

1731 EHCI_OBJS += ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o

1733 HUBD_OBJS += hubd.o

1735 USB_MID_OBJS += usb_mid.o

1737 USB_IA_OBJS += usb_ia.o

1739 UWBA_OBJS += uwba.o uwbai.o

1741 SCSA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o

1743 HWAHC_OBJS += hwahc.o hwahc_util.o

1745 WUSB_DF_OBJS += wusb_df.o
1746 WUSB_FWMOD_OBJS += wusb_fwmod.o

1748 IPF_OBJS += ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1749             ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1750             ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o

1752 IBD_OBJS +=     ibd.o ibd_cm.o

1754 EIBNX_OBJS +=   enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1755                 enx_misc.o enx_q.o enx_ctl.o

1757 EOIB_OBJS +=    eib_adm.o eib_chan.o eib_cmn.o eib_ctl.o eib_data.o \
1758                 eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1759                 eib_rsrc.o eib_svc.o eib_vnic.o

1761 DLPISTUB_OBJS += dlpistub.o

1763 SDP_OBJS +=     sdpddi.o

1765 TRILL_OBJS +=   trill.o

1767 CTF_OBJS += ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1768         ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o

1770 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o

1772 RPCIB_OBJS += rpcib.o

1774 KMDB_OBJS += kdrv.o

1776 AFE_OBJS += afe.o
```



```
1778 BGE_OBJS += bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1779                 bge_atomic.o bge_mii.o bge_send.o bge_recv2.o bge_mii_5906.o

1781 DMFE_OBJS += dmfe_log.o dmfe_main.o dmfe_mii.o

1783 EFE_OBJS += efe.o

1785 ELXL_OBJS += elxl.o

1787 HME_OBJS += hme.o

1789 IXGB_OBJS += ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1790                 ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o

1792 NGE_OBJS += nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1793                 nge_log.o nge_rx.o nge_tx.o nge_xmii.o

1795 PCN_OBJS += pcn.o

1797 RGE_OBJS += rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rxtx.o

1799 URTW_OBJS += urtw.o

1801 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_ani.o arn_phy.o arn_
1802                 arn_main.o arn_recv.o arn_xmit.o arn_rc.o

1804 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o

1806 ATU_OBJS += atu.o

1808 IPW_OBJS += ipw2100_hw.o ipw2100.o

1810 IWI_OBJS += ipw2200_hw.o ipw2200.o

1812 IWH_OBJS += iwh.o

1814 IWK_OBJS += iwk2.o

1816 IWP_OBJS += iwp.o

1818 MWL_OBJS += mwl.o

1820 MWLFW_OBJS += mwlfw_mode.o

1822 WPI_OBJS += wpi.o

1824 RAL_OBJS += rt2560.o ral_rate.o

1826 RUM_OBJS += rum.o

1828 RWD_OBJS += rt2661.o

1830 RWN_OBJS += rt2860.o

1832 UATH_OBJS += uath.o

1834 UATHFW_OBJS += uathfw_mod.o

1836 URAL_OBJS += ural.o

1838 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o

1840 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o

1842 MXFE_OBJS += mxfe.o
```

```
1844 MPTSAS_OBJS += mptsas.o mptsas_impl.o mptsas_init.o mptsas_raid.o mptsas_smhba.o

1846 SFE_OBJS += sfe.o sfe_util.o

1848 BFE_OBJS += bfe.o

1850 BRIDGE_OBJS += bridge.o

1852 IDM_SHARED_OBJS += base64.o

1854 IDM_OBJS +=     $(IDM_SHARED_OBJS) \
1855                 idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o

1857 VR_OBJS += vr.o

1859 ATGE_OBJS += atge_main.o atge_l1e.o atge_mii.o atge_l1.o

1861 YGE_OBJS = yge.o

1863 #
1864 #       Build up defines and paths.
1865 #
1866 LINT_DEFS       += -Dunix

1868 #
1869 #       This duality can be removed when the native and target compilers
1870 #       are the same (or at least recognize the same command line syntax!)
1871 #       It is a bug in the current compilation system that the assember
1872 #       can't process the -Y I, flag.
1873 #
1874 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1875 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1876 INCLUDE_PATH    += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common

1878 PCIEB_OBJS += pcieb.o

1880 #       Chelsio N110 10G NIC driver module
1881 #
1882 CH_OBJS = ch.o glue.o pe.o sge.o

1884 CH_COM_OBJS =   ch_mac.o ch_subr.o cspi.o espi.o ixf1010.o mc3.o mc4.o mc5.o \
1885                 mv88e1xxx.o mv88x201x.o my3126.o pm3393.o tp.o ulp.o \
1886                 vsc7321.o vsc7326.o xpak.o

1888 #
1889 #       PCI strings file
1890 #
1891 PCI_STRING_OBJS = pci_strings.o

1893 NET_DACF_OBJS += net_dacf.o

1895 #
1896 #       Xframe 10G NIC driver module
1897 #
1898 XGE_OBJS = xge.o xgell.o

1900 XGE_HAL_OBJS =  xgehal-channel.o xgehal-fifo.o xgehal-ring.o  xgehal-config.o \
1901                 xgehal-driver.o  xgehal-mm.o xgehal-stats.o  xgehal-device.o \
1902                 xge-queue.o  xgehal-mgmt.o xgehal-mgmtaux.o

1904 #
1905 #       e1000g module
1906 #
1907 E1000G_OBJS +=  e1000_80003es2lan.o e1000_82540.o e1000_82541.o e1000_82542.o \
1908                 e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
```



```
1909                 e1000_mac.o e1000_manage.o e1000_nvm.o e1000_osdep.o \
1910                 e1000_phy.o e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1911                 e1000g_tx.o e1000g_rx.o e1000g_stat.o

1913 #
1914 #       Intel 82575 1G NIC driver module
1915 #
1916 IGB_OBJS =      igb_82575.o igb_api.o igb_mac.o igb_manage.o \
1917                 igb_nvm.o igb_osdep.o igb_phy.o igb_buf.o \
1918                 igb_debug.o igb_gld.o igb_log.o igb_main.o \
1919                 igb_rx.o igb_stat.o igb_tx.o

1921 #
1922 #       Intel Pro/100 NIC driver module
1923 #
1924 IPRB_OBJS =     iprb.o

1926 #
1927 #       Intel 10GbE PCIE NIC driver module
1928 #
1929 IXGBE_OBJS =    ixgbe_82598.o ixgbe_82599.o ixgbe_api.o         \
1930                 ixgbe_common.o ixgbe_phy.o                      \
1931                 ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o           \
1932                 ixgbe_log.o ixgbe_main.o                        \
1933                 ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o           \
1934                 ixgbe_tx.o

1936 #
1937 #       NIU 10G/1G driver module
1938 #
1939 NXGE_OBJS =     nxge_mac.o nxge_ipp.o nxge_rxdma.o              \
1940                 nxge_txdma.o nxge_txc.o nxge_main.o             \
1941                 nxge_hw.o nxge_fzc.o nxge_virtual.o             \
1942                 nxge_send.o nxge_classify.o nxge_fflp.o         \
1943                 nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o       \
1944                 nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o      \
1945                 nxge_hio.o nxge_hio_guest.o nxge_intr.o

1947 NXGE_NPI_OBJS = \
1948                 npi.o npi_mac.o npi_ipp.o                       \
1949                 npi_txdma.o npi_rxdma.o npi_txc.o               \
1950                 npi_zcp.o npi_espc.o npi_fflp.o                 \
1951                 npi_vir.o

1953 NXGE_HCALL_OBJS =       \
1954                 nxge_hcall.o

1956 #
1957 #       kiconv modules
1958 #
1959 KICONV_EMEA_OBJS += kiconv_emea.o

1961 KICONV_JA_OBJS += kiconv_ja.o

1963 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o

1965 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o

1967 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o

1969 #
1970 #       AAC module
1971 #
1972 AAC_OBJS = aac.o aac_ioctl.o

1974 #
```

```
#       sdcard modules
#
SDA_OBJS =      sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o
SDHOST_OBJS =   sdhost.o

#
#       hxge 10G driver module
#
HXGE_OBJS =     hxge_main.o hxge_vmac.o hxge_send.o             \
                hxge_txdma.o hxge_rxdma.o hxge_virtual.o        \
                hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o    \
                hxge_ndd.o hxge_pfc.o                           \
                hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o        \
                hpi_vir.o hpi_pfc.o

#
#       MEGARAID_SAS module
#
MEGA_SAS_OBJS = megaraid_sas.o

#
#       MR_SAS module
#
MR_SAS_OBJS = mr_sas.o

#
#       ISCSI_INITIATOR module
#
ISCSI_INITIATOR_OBJS =  chap.o iscsi_io.o iscsi_thread.o        \
                        iscsi_ioctl.o iscsid.o iscsi.o          \
                        iscsi_login.o isns_client.o iscsiAuthClient.o   \
                        iscsi_lun.o iscsiAuthClientGlue.o       \
                        iscsi_net.o nvfile.o iscsi_cmd.o        \
                        iscsi_queue.o persistent.o iscsi_conn.o \
                        iscsi_sess.o radius_auth.o iscsi_crc.o  \
                        iscsi_stats.o radius_packet.o iscsi_doorclt.o   \
                        iscsi_targetparam.o utils.o kifconf.o

#
#       ntxn 10Gb/1Gb NIC driver module
#
NTXN_OBJS =     unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
                        unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

#
#       Myricom 10Gb NIC driver module
#
MYRI10GE_OBJS = myri10ge.o myri10ge_lro.o

#       nulldriver module
#
NULLDRIVER_OBJS =       nulldriver.o

TPM_OBJS =      tpm.o tpm_hcall.o
```

```
**********************************************************
   72399 Mon Jul  9 14:38:11 2012
new/usr/src/uts/common/Makefile.rules
dccp: starting module template
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
#

#
# Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
#

#
# uts/common/Makefile.rules
#
#       This Makefile defines all the file build rules for the directory
# uts/common and its children. These are the source files which may
# be considered common to all SunOS systems.
#
#       The following two-level ordering must be maintained in this file.
#         Lines are sorted first in order of decreasing specificity based on
#         the first directory component.  That is, sun4u rules come before
#         sparc rules come before common rules.
#
#         Lines whose initial directory components are equal are sorted
#         alphabetically by the remaining components.

#
#       Section 1a: C objects build rules
#
$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/aes/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/arcfour/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/blowfish/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/ecc/%.c
        $(COMPILE.c) -o $@ $<
```



```
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/modes/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/padding/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/rng/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/rsa/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/bignum/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/bignum/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/mpi/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/acl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/avl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/ucode/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/brand/sn1/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/brand/solaris10/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/c2/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/conf/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/contract/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/cpr/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)
```

```
128 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ctf/%.c
129         $(COMPILE.c) -o $@ $<
130         $(CTFCONVERT_O)

132 $(OBJS_DIR)/%.o:                $(COMMONBASE)/ctf/%.c
133         $(COMPILE.c) -o $@ $<
134         $(CTFCONVERT_O)

136 $(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/des/%.c
137         $(COMPILE.c) -o $@ $<
138         $(CTFCONVERT_O)

140 $(OBJS_DIR)/%.o:                $(COMMONBASE)/smbios/%.c
141         $(COMPILE.c) -o $@ $<
142         $(CTFCONVERT_O)

144 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/des/%.c
145         $(COMPILE.c) -o $@ $<
146         $(CTFCONVERT_O)

148 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/crypto/api/%.c
149         $(COMPILE.c) -o $@ $<
150         $(CTFCONVERT_O)

152 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/crypto/core/%.c
153         $(COMPILE.c) -o $@ $<
154         $(CTFCONVERT_O)

156 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/crypto/io/%.c
157         $(COMPILE.c) -o $@ $<
158         $(CTFCONVERT_O)

160 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/crypto/spi/%.c
161         $(COMPILE.c) -o $@ $<
162         $(CTFCONVERT_O)

164 $(OBJS_DIR)/%.o:                $(COMMONBASE)/pci/%.c
165         $(COMPILE.c) -o $@ $<
166         $(CTFCONVERT_O)

168 $(OBJS_DIR)/%.o:                $(COMMONBASE)/devid/%.c
169         $(COMPILE.c) -o $@ $<
170         $(CTFCONVERT_O)

172 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/disp/%.c
173         $(COMPILE.c) -o $@ $<
174         $(CTFCONVERT_O)

176 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/dtrace/%.c
177         $(COMPILE.c) -o $@ $<
178         $(CTFCONVERT_O)

180 $(OBJS_DIR)/%.o:                $(COMMONBASE)/exacct/%.c
181         $(COMPILE.c) -o $@ $<
182         $(CTFCONVERT_O)

184 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/exec/aout/%.c
185         $(COMPILE.c) -o $@ $<
186         $(CTFCONVERT_O)

188 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/exec/elf/%.c
189         $(COMPILE.c) -o $@ $<
190         $(CTFCONVERT_O)

192 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/exec/intp/%.c
193         $(COMPILE.c) -o $@ $<
```



```
194         $(CTFCONVERT_O)

196 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/exec/shbin/%.c
197         $(COMPILE.c) -o $@ $<
198         $(CTFCONVERT_O)

200 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/exec/java/%.c
201         $(COMPILE.c) -o $@ $<
202         $(CTFCONVERT_O)

204 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/%.c
205         $(COMPILE.c) -o $@ $<
206         $(CTFCONVERT_O)

208 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/autofs/%.c
209         $(COMPILE.c) -o $@ $<
210         $(CTFCONVERT_O)

212 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/cachefs/%.c
213         $(COMPILE.c) -o $@ $<
214         $(CTFCONVERT_O)

216 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/dcfs/%.c
217         $(COMPILE.c) -o $@ $<
218         $(CTFCONVERT_O)

220 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/devfs/%.c
221         $(COMPILE.c) -o $@ $<
222         $(CTFCONVERT_O)

224 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/ctfs/%.c
225         $(COMPILE.c) -o $@ $<
226         $(CTFCONVERT_O)

228 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/doorfs/%.c
229         $(COMPILE.c) -o $@ $<
230         $(CTFCONVERT_O)

232 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/dev/%.c
233         $(COMPILE.c) -o $@ $<
234         $(CTFCONVERT_O)

236 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/fd/%.c
237         $(COMPILE.c) -o $@ $<
238         $(CTFCONVERT_O)

240 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/fifofs/%.c
241         $(COMPILE.c) -o $@ $<
242         $(CTFCONVERT_O)

244 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/hsfs/%.c
245         $(COMPILE.c) -o $@ $<
246         $(CTFCONVERT_O)

248 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/lofs/%.c
249         $(COMPILE.c) -o $@ $<
250         $(CTFCONVERT_O)

252 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/mntfs/%.c
253         $(COMPILE.c) -o $@ $<
254         $(CTFCONVERT_O)

256 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/namefs/%.c
257         $(COMPILE.c) -o $@ $<
258         $(CTFCONVERT_O)
```

```
260 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/nfs/%.c
261         $(COMPILE.c) -o $@ $<
262         $(CTFCONVERT_O)

264 $(OBJS_DIR)/%.o:                $(COMMONBASE)/smbsrv/%.c
265         $(COMPILE.c) -o $@ $<
266         $(CTFCONVERT_O)

268 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/smbsrv/%.c
269         $(COMPILE.c) -o $@ $<
270         $(CTFCONVERT_O)

272 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/objfs/%.c
273         $(COMPILE.c) -o $@ $<
274         $(CTFCONVERT_O)

276 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/pcfs/%.c
277         $(COMPILE.c) -o $@ $<
278         $(CTFCONVERT_O)

280 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/portfs/%.c
281         $(COMPILE.c) -o $@ $<
282         $(CTFCONVERT_O)

284 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/proc/%.c
285         $(COMPILE.c) -o $@ $<
286         $(CTFCONVERT_O)

288 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/sharefs/%.c
289         $(COMPILE.c) -o $@ $<
290         $(CTFCONVERT_O)

292 $(OBJS_DIR)/%.o:                $(COMMONBASE)/smbclnt/%.c
293         $(COMPILE.c) -o $@ $<
294         $(CTFCONVERT_O)

296 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/smbclnt/netsmb/%.c
297         $(COMPILE.c) -o $@ $<
298         $(CTFCONVERT_O)

300 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/smbclnt/smbfs/%.c
301         $(COMPILE.c) -o $@ $<
302         $(CTFCONVERT_O)

304 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/sockfs/%.c
305         $(COMPILE.c) -o $@ $<
306         $(CTFCONVERT_O)

308 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/specfs/%.c
309         $(COMPILE.c) -o $@ $<
310         $(CTFCONVERT_O)

312 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/swapfs/%.c
313         $(COMPILE.c) -o $@ $<
314         $(CTFCONVERT_O)

316 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/tmpfs/%.c
317         $(COMPILE.c) -o $@ $<
318         $(CTFCONVERT_O)

320 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/udfs/%.c
321         $(COMPILE.c) -o $@ $<
322         $(CTFCONVERT_O)

324 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/ufs/%.c
325         $(COMPILE.c) -o $@ $<
```



```
326         $(CTFCONVERT_O)

328 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/vscan/%.c
329         $(COMPILE.c) -o $@ $<
330         $(CTFCONVERT_O)

332 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/zfs/%.c
333         $(COMPILE.c) -o $@ $<
334         $(CTFCONVERT_O)

336 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/fs/zut/%.c
337         $(COMPILE.c) -o $@ $<
338         $(CTFCONVERT_O)

340 $(OBJS_DIR)/%.o:                $(COMMONBASE)/xattr/%.c
341         $(COMPILE.c) -o $@ $<
342         $(CTFCONVERT_O)

344 $(OBJS_DIR)/%.o:                $(COMMONBASE)/zfs/%.c
345         $(COMPILE.c) -o $@ $<
346         $(CTFCONVERT_O)

348 $(OBJS_DIR)/%.o:        $(UTSBASE)/common/io/scsi/adapters/pmcs/%.c
349         $(COMPILE.c) -o $@ $<
350         $(CTFCONVERT_O)

352 $(OBJS_DIR)/%.o:        $(UTSBASE)/common/io/scsi/adapters/pmcs/%.bin
353         $(COMPILE.b) -o $@ $<
354         $(CTFCONVERT_O)

356 $(OBJS_DIR)/%.o:                $(COMMONBASE)/fsreparse/%.c
357         $(COMPILE.c) -o $@ $<
358         $(CTFCONVERT_O)

360 KMECHKRB5_BASE=$(UTSBASE)/common/gssapi/mechs/krb5

362 KGSSDFLAGS=-I $(UTSBASE)/common/gssapi/include

364 # Note, KRB5_DEFS can be assigned various preprocessor flags,
365 # typically -D defines on the make invocation.  The standard compiler
366 # flags will not be overwritten.
367 KGSSDFLAGS += $(KRB5_DEFS)

369 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/gssapi/%.c
370         $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
371         $(CTFCONVERT_O)

373 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/gssapi/mechs/dummy/%.c
374         $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
375         $(CTFCONVERT_O)

377 $(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/%.c
378         $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
379         $(CTFCONVERT_O)

381 $(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/%.c
382         $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
383         $(CTFCONVERT_O)

385 $(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/des/%.c
386         $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
387         $(CTFCONVERT_O)

389 $(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/arcfour/%.c
390         $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
391         $(CTFCONVERT_O)
```

```
$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/dk/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/enc_provider/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/hash_provider/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/keyhash_provider/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/raw/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/crypto/old/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/krb5/krb/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/krb5/os/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/ser_sctx.o := CPPFLAGS += -DPROVIDE_KERNEL_IMPORT=1

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/mech/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(KMECHKRB5_BASE)/profile/%.c
        $(COMPILE.c) $(KGSSDFLAGS) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ncall/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ns/dsw/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ns/nsctl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ns/rdc/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ns/sdbc/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ns/solaris/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)
```



```
$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ns/sv/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/avs/ns/unistat/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/idmap/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/arp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/dccp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

#endif /* ! codereview */
$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/ip/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/ipnet/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/iptun/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/kssl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/sctp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/tcp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/ilb/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/ipf/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/net/patricia/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/udp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)
```

```
524 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/nca/%.c
525         $(COMPILE.c) -o $@ $<
526         $(CTFCONVERT_O)
527
528 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/sockmods/%.c
529         $(COMPILE.c) -o $@ $<
530         $(CTFCONVERT_O)
531
532 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/dlpistub/%.c
533         $(COMPILE.c) -o $@ $<
534         $(CTFCONVERT_O)
535
536 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/%.c
537         $(COMPILE.c) -o $@ $<
538         $(CTFCONVERT_O)
539
540 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/1394/%.c
541         $(COMPILE.c) -o $@ $<
542         $(CTFCONVERT_O)
543
544 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/1394/adapters/%.c
545         $(COMPILE.c) -o $@ $<
546         $(CTFCONVERT_O)
547
548 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/1394/targets/av1394/%.c
549         $(COMPILE.c) -o $@ $<
550         $(CTFCONVERT_O)
551
552 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/1394/targets/dcam1394/%.c
553         $(COMPILE.c) -o $@ $<
554         $(CTFCONVERT_O)
555
556 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/1394/targets/scsa1394/%.c
557         $(COMPILE.c) -o $@ $<
558         $(CTFCONVERT_O)
559
560 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sbp2/%.c
561         $(COMPILE.c) -o $@ $<
562         $(CTFCONVERT_O)
563
564 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/aac/%.c
565         $(COMPILE.c) -o $@ $<
566         $(CTFCONVERT_O)
567
568 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/afe/%.c
569         $(COMPILE.c) -o $@ $<
570         $(CTFCONVERT_O)
571
572 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/atge/%.c
573         $(COMPILE.c) -o $@ $<
574         $(CTFCONVERT_O)
575
576 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/arn/%.c
577         $(COMPILE.c) -o $@ $<
578         $(CTFCONVERT_O)
579
580 $(OBJS_DIR)/%.o:              $(UTSBASE)/common/io/ath/%.c
581         $(COMPILE.c) -o $@ $<
582         $(CTFCONVERT_O)
583
584 $(OBJS_DIR)/%.o:              $(UTSBASE)/common/io/atu/%.c
585         $(COMPILE.c) -o $@ $<
586         $(CTFCONVERT_O)
587
588 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/impl/%.c
589         $(COMPILE.c) -o $@ $<
```



```
590         $(CTFCONVERT_O)
591
592 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/ac97/%.c
593         $(COMPILE.c) -o $@ $<
594         $(CTFCONVERT_O)
595
596 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audioens/%.c
597         $(COMPILE.c) -o $@ $<
598         $(CTFCONVERT_O)
599
600 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audioemu10k/%.c
601         $(COMPILE.c) -o $@ $<
602         $(CTFCONVERT_O)
603
604 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audio1575/%.c
605         $(COMPILE.c) -o $@ $<
606         $(CTFCONVERT_O)
607
608 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audio810/%.c
609         $(COMPILE.c) -o $@ $<
610         $(CTFCONVERT_O)
611
612 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiocmi/%.c
613         $(COMPILE.c) -o $@ $<
614         $(CTFCONVERT_O)
615
616 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiocmihd/%.c
617         $(COMPILE.c) -o $@ $<
618         $(CTFCONVERT_O)
619
620 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiohd/%.c
621         $(COMPILE.c) -o $@ $<
622         $(CTFCONVERT_O)
623
624 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audioixp/%.c
625         $(COMPILE.c) -o $@ $<
626         $(CTFCONVERT_O)
627
628 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiols/%.c
629         $(COMPILE.c) -o $@ $<
630         $(CTFCONVERT_O)
631
632 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiopci/%.c
633         $(COMPILE.c) -o $@ $<
634         $(CTFCONVERT_O)
635
636 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiop16x/%.c
637         $(COMPILE.c) -o $@ $<
638         $(CTFCONVERT_O)
639
640 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiosolo/%.c
641         $(COMPILE.c) -o $@ $<
642         $(CTFCONVERT_O)
643
644 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiots/%.c
645         $(COMPILE.c) -o $@ $<
646         $(CTFCONVERT_O)
647
648 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiovia823x/%.c
649         $(COMPILE.c) -o $@ $<
650         $(CTFCONVERT_O)
651
652 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/audio/drv/audiovia97/%.c
653         $(COMPILE.c) -o $@ $<
654         $(CTFCONVERT_O)
```

```
$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/bfe/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/bge/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/blkdev/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/bpf/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/cardbus/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/stmf/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/port/fct/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/port/qlt/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/port/srpt/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/port/fcoet/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/iscsit/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/port/iscsit/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/port/pppt/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/comstar/lu/stmf_sbd/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/dld/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/dls/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/dmfe/%.c
        $(COMPILE.c) -o $@ $<
```



```
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/drm/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/efe/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/elxl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fcoe/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/hme/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/pciex/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/hotplug/hpcsvc/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/pciex/hotplug/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/hotplug/pcihp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/rds/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/rdsv3/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/iser/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/ibd/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/eoib/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/of/sol_ofs/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/of/sol_ucma/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)
```

```
$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/of/sol_umad/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/of/sol_uverbs/%.
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/sdp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/mgt/ibcm/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/mgt/ibdm/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/mgt/ibdma/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/mgt/ibmf/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/ibnex/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/ibtl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/adapters/tavor/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/adapters/hermon/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ib/clients/daplt/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(COMMONBASE)/iscsi/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/idm/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ipw/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/iwh/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/iwi/%.c
        $(COMPILE.c) -o $@ $<
```



```
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/iwk/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/iwp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/kb8042/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/kbtrans/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ksocket/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/aggr/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/hotspares/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/md/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/mirror/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/notify/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/raid/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/softpart/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/stripe/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/lvm/trans/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mac/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)
```

```
$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mac/plugins/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mega_sas/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mii/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mr_sas/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/adapters/mpt_sas/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mxfe/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mwl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/mwl/mwl_fw/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/net80211/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/nge/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/nxge/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/nxge/npi/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/nxge/%.s
        $(COMPILE.s) -o $@ $<

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/pci-ide/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/pcmcia/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/pcan/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/pcn/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)
```



```
$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/pcwl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ppp/sppp/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ppp/spppasyn/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ppp/spppstun/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ral/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/rge/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/rtls/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/rsm/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/rtw/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/rum/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/rwd/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/rwn/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sata/adapters/ahci/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sata/adapters/nv_sata/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sata/adapters/si3124/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sata/impl/%.c
        $(COMPILE.c) -o $@ $<
        $(CTFCONVERT_O)

$(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/conf/%.c
```

```
1052         $(COMPILE.c) -o $@ $<
1053         $(CTFCONVERT_O)

1055 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/impl/%.c
1056         $(COMPILE.c) -o $@ $<
1057         $(CTFCONVERT_O)

1059 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/targets/%.c
1060         $(COMPILE.c) -o $@ $<
1061         $(CTFCONVERT_O)

1063 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/adapters/%.c
1064         $(COMPILE.c) -o $@ $<
1065         $(CTFCONVERT_O)

1067 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/adapters/blk2scsa/%.c
1068         $(COMPILE.c) -o $@ $<
1069         $(CTFCONVERT_O)

1071 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/%.c
1072         $(COMPILE.c) -o $@ $<
1073         $(CTFCONVERT_O)

1075 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/fop
1076         $(COMPILE.c) -o $@ $<
1077         $(CTFCONVERT_O)

1079 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fibre-channel/ulp/%.c
1080         $(COMPILE.c) -o $@ $<
1081         $(CTFCONVERT_O)

1083 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fibre-channel/impl/%.c
1084         $(COMPILE.c) -o $@ $<
1085         $(CTFCONVERT_O)

1087 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fibre-channel/fca/qlc/%.c
1088         $(COMPILE.c) -o $@ $<
1089         $(CTFCONVERT_O)

1091 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fibre-channel/fca/qlge/%.c
1092         $(COMPILE.c) -o $@ $<
1093         $(CTFCONVERT_O)

1095 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fibre-channel/fca/emlxs/%.c
1096         $(COMPILE.c) -o $@ $<
1097         $(CTFCONVERT_O)

1099 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fibre-channel/fca/oce/%.c
1100         $(COMPILE.c) -o $@ $<
1101         $(CTFCONVERT_O)

1103 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/fibre-channel/fca/fcoei/%.c
1104         $(COMPILE.c) -o $@ $<
1105         $(CTFCONVERT_O)

1107 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sdcard/adapters/sdhost/%.c
1108         $(COMPILE.c) -o $@ $<
1109         $(CTFCONVERT_O)

1111 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sdcard/impl/%.c
1112         $(COMPILE.c) -o $@ $<
1113         $(CTFCONVERT_O)

1115 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sdcard/targets/sdcard/%.c
1116         $(COMPILE.c) -o $@ $<
1117         $(CTFCONVERT_O)
```



```
1119 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/sfe/%.c
1120         $(COMPILE.c) -o $@ $<
1121         $(CTFCONVERT_O)

1123 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/simnet/%.c
1124         $(COMPILE.c) -o $@ $<
1125         $(CTFCONVERT_O)

1127 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/softmac/%.c
1128         $(COMPILE.c) -o $@ $<
1129         $(CTFCONVERT_O)

1131 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/uath/%.c
1132         $(COMPILE.c) -o $@ $<
1133         $(CTFCONVERT_O)

1135 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/uath/uath_fw/%.c
1136         $(COMPILE.c) -o $@ $<
1137         $(CTFCONVERT_O)

1139 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ural/%.c
1140         $(COMPILE.c) -o $@ $<
1141         $(CTFCONVERT_O)

1143 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/urtw/%.c
1144         $(COMPILE.c) -o $@ $<
1145         $(CTFCONVERT_O)

1147 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/audio/usb_ac/%.
1148         $(COMPILE.c) -o $@ $<
1149         $(CTFCONVERT_O)

1151 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/audio/usb_as/%.
1152         $(COMPILE.c) -o $@ $<
1153         $(CTFCONVERT_O)

1155 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/audio/usb_ah/%.
1156         $(COMPILE.c) -o $@ $<
1157         $(CTFCONVERT_O)

1159 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbskel/%.c
1160         $(COMPILE.c) -o $@ $<
1161         $(CTFCONVERT_O)

1163 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/video/usbvc/%.c
1164         $(COMPILE.c) -o $@ $<
1165         $(CTFCONVERT_O)

1167 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/hwarc/%.c
1168         $(COMPILE.c) -o $@ $<
1169         $(CTFCONVERT_O)

1171 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/hid/%.c
1172         $(COMPILE.c) -o $@ $<
1173         $(CTFCONVERT_O)

1175 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/hidparser/%.c
1176         $(COMPILE.c) -o $@ $<
1177         $(CTFCONVERT_O)

1179 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/printer/%.c
1180         $(COMPILE.c) -o $@ $<
1181         $(CTFCONVERT_O)

1183 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbkbm/%.c
```

```
1184         $(COMPILE.c) -o $@ $<
1185         $(CTFCONVERT_O)

1187 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbms/%.c
1188         $(COMPILE.c) -o $@ $<
1189         $(CTFCONVERT_O)

1191 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbinput/usbwcm
1192         $(COMPILE.c) -o $@ $<
1193         $(CTFCONVERT_O)

1195 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/ugen/%.c
1196         $(COMPILE.c) -o $@ $<
1197         $(CTFCONVERT_O)

1199 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbser/%.c
1200         $(COMPILE.c) -o $@ $<
1201         $(CTFCONVERT_O)

1203 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbser/usbsacm/
1204         $(COMPILE.c) -o $@ $<
1205         $(CTFCONVERT_O)

1207 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbser/usbftdi/
1208         $(COMPILE.c) -o $@ $<
1209         $(CTFCONVERT_O)

1211 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbser/usbser_k
1212         $(COMPILE.c) -o $@ $<
1213         $(CTFCONVERT_O)

1215 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbser/usbsprl/
1216         $(COMPILE.c) -o $@ $<
1217         $(CTFCONVERT_O)

1219 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/wusb_df/%.c
1220         $(COMPILE.c) -o $@ $<
1221         $(CTFCONVERT_O)

1223 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/hwa1480_fw/%.c
1224         $(COMPILE.c) -o $@ $<
1225         $(CTFCONVERT_O)

1227 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/wusb_ca/%.c
1228         $(COMPILE.c) -o $@ $<
1229         $(CTFCONVERT_O)

1231 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/clients/usbecm/%.c
1232         $(COMPILE.c) -o $@ $<
1233         $(CTFCONVERT_O)

1235 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/hcd/openhci/%.c
1236         $(COMPILE.c) -o $@ $<
1237         $(CTFCONVERT_O)

1239 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/hcd/ehci/%.c
1240         $(COMPILE.c) -o $@ $<
1241         $(CTFCONVERT_O)

1243 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/hcd/uhci/%.c
1244         $(COMPILE.c) -I../../common -o $@ $<
1245         $(CTFCONVERT_O)

1247 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/hubd/%.c
1248         $(COMPILE.c) -o $@ $<
1249         $(CTFCONVERT_O)
```



```
1251 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/scsa2usb/%.c
1252         $(COMPILE.c) -o $@ $<
1253         $(CTFCONVERT_O)

1255 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/usb_mid/%.c
1256         $(COMPILE.c) -o $@ $<
1257         $(CTFCONVERT_O)

1259 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/usb_ia/%.c
1260         $(COMPILE.c) -o $@ $<
1261         $(CTFCONVERT_O)

1263 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/usba/%.c
1264         $(COMPILE.c) -o $@ $<
1265         $(CTFCONVERT_O)

1267 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/usba10/%.c
1268         $(COMPILE.c) -o $@ $<
1269         $(CTFCONVERT_O)

1271 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/usb/hwa/hwahc/%.c
1272         $(COMPILE.c) -o $@ $<
1273         $(CTFCONVERT_O)

1275 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/uwb/uwba/%.c
1276         $(COMPILE.c) -o $@ $<
1277         $(CTFCONVERT_O)

1279 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/vuidmice/%.c
1280         $(COMPILE.c) -o $@ $<
1281         $(CTFCONVERT_O)

1283 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/vnic/%.c
1284         $(COMPILE.c) -o $@ $<
1285         $(CTFCONVERT_O)

1287 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/wpi/%.c
1288         $(COMPILE.c) -o $@ $<
1289         $(CTFCONVERT_O)

1291 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/zyd/%.c
1292         $(COMPILE.c) -o $@ $<
1293         $(CTFCONVERT_O)

1295 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/chxge/com/%.c
1296         $(COMPILE.c) -o $@ $<
1297         $(CTFCONVERT_O)

1299 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/chxge/%.c
1300         $(COMPILE.c) -o $@ $<
1301         $(CTFCONVERT_O)

1303 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ixgb/%.c
1304         $(COMPILE.c) -o $@ $<
1305         $(CTFCONVERT_O)

1307 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/xge/drv/%.c
1308         $(COMPILE.c) -o $@ $<
1309         $(CTFCONVERT_O)

1311 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/xge/hal/xgehal/%.c
1312         $(COMPILE.c) -o $@ $<
1313         $(CTFCONVERT_O)

1315 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/e1000g/%.c
```

```
1316         $(COMPILE.c) -o $@ $<
1317         $(CTFCONVERT_O)

1319 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/igb/%.c
1320         $(COMPILE.c) -o $@ $<
1321         $(CTFCONVERT_O)

1323 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/iprb/%.c
1324         $(COMPILE.c) -o $@ $<
1325         $(CTFCONVERT_O)

1327 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ixgbe/%.c
1328         $(COMPILE.c) -o $@ $<
1329         $(CTFCONVERT_O)

1331 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/ntxn/%.c
1332         $(COMPILE.c) -o $@ $<
1333         $(CTFCONVERT_O)

1335 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/myri10ge/drv/%.c
1336         $(COMPILE.c) -o $@ $<
1337         $(CTFCONVERT_O)

1339 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ipp/%.c
1340         $(COMPILE.c) -o $@ $<
1341         $(CTFCONVERT_O)

1343 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ipp/ipgpc/%.c
1344         $(COMPILE.c) -o $@ $<
1345         $(CTFCONVERT_O)

1347 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ipp/dlcosmk/%.c
1348         $(COMPILE.c) -o $@ $<
1349         $(CTFCONVERT_O)

1351 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ipp/flowacct/%.c
1352         $(COMPILE.c) -o $@ $<
1353         $(CTFCONVERT_O)

1355 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ipp/dscpmk/%.c
1356         $(COMPILE.c) -o $@ $<
1357         $(CTFCONVERT_O)

1359 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ipp/meters/%.c
1360         $(COMPILE.c) -o $@ $<
1361         $(CTFCONVERT_O)

1363 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/kiconv/kiconv_emea/%.c
1364         $(COMPILE.c) -o $@ $<
1365         $(CTFCONVERT_O)

1367 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/kiconv/kiconv_ja/%.c
1368         $(COMPILE.c) -o $@ $<
1369         $(CTFCONVERT_O)

1371 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/kiconv/kiconv_ko/%.c
1372         $(COMPILE.c) -o $@ $<
1373         $(CTFCONVERT_O)

1375 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/kiconv/kiconv_sc/%.c
1376         $(COMPILE.c) -o $@ $<
1377         $(CTFCONVERT_O)

1379 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/kiconv/kiconv_tc/%.c
1380         $(COMPILE.c) -o $@ $<
1381         $(CTFCONVERT_O)
```



```
1383 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/kmdb/%.c
1384         $(COMPILE.c) -o $@ $<
1385         $(CTFCONVERT_O)

1387 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/ktli/%.c
1388         $(COMPILE.c) -o $@ $<
1389         $(CTFCONVERT_O)

1391 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/scsi/adapters/iscsi/%.c
1392         $(COMPILE.c) -o $@ $<
1393         $(CTFCONVERT_O)

1395 $(OBJS_DIR)/%.o:                $(COMMONBASE)/iscsi/%.c
1396         $(COMPILE.c) -o $@ $<
1397         $(CTFCONVERT_O)

1399 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/inet/kifconf/%.c
1400         $(COMPILE.c) -o $@ $<
1401         $(CTFCONVERT_O)

1403 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/vr/%.c
1404         $(COMPILE.c) -o $@ $<
1405         $(CTFCONVERT_O)

1407 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/yge/%.c
1408         $(COMPILE.c) -o $@ $<
1409         $(CTFCONVERT_O)

1411 #
1412 # krtld must refer to its own bzero/bcopy until the kernel is fully linked
1413 #
1414 $(OBJS_DIR)/bootrd.o            := CPPFLAGS += -DKOBJ_OVERRIDES
1415 $(OBJS_DIR)/doreloc.o           := CPPFLAGS += -DKOBJ_OVERRIDES
1416 $(OBJS_DIR)/kobj.o              := CPPFLAGS += -DKOBJ_OVERRIDES
1417 $(OBJS_DIR)/kobj_boot.o         := CPPFLAGS += -DKOBJ_OVERRIDES
1418 $(OBJS_DIR)/kobj_bootflags.o    := CPPFLAGS += -DKOBJ_OVERRIDES
1419 $(OBJS_DIR)/kobj_convrelstr.o   := CPPFLAGS += -DKOBJ_OVERRIDES
1420 $(OBJS_DIR)/kobj_isa.o          := CPPFLAGS += -DKOBJ_OVERRIDES
1421 $(OBJS_DIR)/kobj_kdi.o          := CPPFLAGS += -DKOBJ_OVERRIDES
1422 $(OBJS_DIR)/kobj_lm.o           := CPPFLAGS += -DKOBJ_OVERRIDES
1423 $(OBJS_DIR)/kobj_reloc.o        := CPPFLAGS += -DKOBJ_OVERRIDES
1424 $(OBJS_DIR)/kobj_stubs.o        := CPPFLAGS += -DKOBJ_OVERRIDES
1425 $(OBJS_DIR)/kobj_subr.o         := CPPFLAGS += -DKOBJ_OVERRIDES

1427 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/krtld/%.c
1428         $(COMPILE.c) -o $@ $<
1429         $(CTFCONVERT_O)

1431 $(OBJS_DIR)/%.o:                $(COMMONBASE)/list/%.c
1432         $(COMPILE.c) -o $@ $<
1433         $(CTFCONVERT_O)

1435 $(OBJS_DIR)/%.o:                $(COMMONBASE)/lvm/%.c
1436         $(COMPILE.c) -o $@ $<
1437         $(CTFCONVERT_O)

1439 $(OBJS_DIR)/%.o:                $(COMMONBASE)/lzma/%.c
1440         $(COMPILE.c) -o $@ $<
1441         $(CTFCONVERT_O)

1443 $(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/md4/%.c
1444         $(COMPILE.c) -o $@ $<
1445         $(CTFCONVERT_O)

1447 $(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/md5/%.c
```

```
1448         $(COMPILE.c) -o $@ $<
1449         $(CTFCONVERT_O)

1451 $(OBJS_DIR)/%.o:                $(COMMONBASE)/net/dhcp/%.c
1452         $(COMPILE.c) -o $@ $<
1453         $(CTFCONVERT_O)

1455 $(OBJS_DIR)/%.o:                $(COMMONBASE)/nvpair/%.c
1456         $(COMPILE.c) -o $@ $<
1457         $(CTFCONVERT_O)

1459 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/os/%.c
1460         $(COMPILE.c) -o $@ $<
1461         $(CTFCONVERT_O)

1463 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/pcmcia/cis/%.c
1464         $(COMPILE.c) -o $@ $<
1465         $(CTFCONVERT_O)

1467 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/pcmcia/cs/%.c
1468         $(COMPILE.c) -o $@ $<
1469         $(CTFCONVERT_O)

1471 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/pcmcia/nexus/%.c
1472         $(COMPILE.c) -o $@ $<
1473         $(CTFCONVERT_O)

1475 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/pcmcia/pcs/%.c
1476         $(COMPILE.c) -o $@ $<
1477         $(CTFCONVERT_O)

1479 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/rpc/%.c
1480         $(COMPILE.c) -o $@ $<
1481         $(CTFCONVERT_O)

1483 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/rpc/sec/%.c
1484         $(COMPILE.c) -o $@ $<
1485         $(CTFCONVERT_O)

1487 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/rpc/sec_gss/%.c
1488         $(COMPILE.c) -o $@ $<
1489         $(CTFCONVERT_O)

1491 $(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/sha1/%.c
1492         $(COMPILE.c) -o $@ $<
1493         $(CTFCONVERT_O)

1495 $(OBJS_DIR)/%.o:                $(COMMONBASE)/crypto/sha2/%.c
1496         $(COMPILE.c) -o $@ $<
1497         $(CTFCONVERT_O)

1499 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/syscall/%.c
1500         $(COMPILE.c) -o $@ $<
1501         $(CTFCONVERT_O)

1503 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/tnf/%.c
1504         $(COMPILE.c) -o $@ $<
1505         $(CTFCONVERT_O)

1507 $(OBJS_DIR)/%.o:                $(COMMONBASE)/tsol/%.c
1508         $(COMPILE.c) -o $@ $<
1509         $(CTFCONVERT_O)

1511 $(OBJS_DIR)/%.o:                $(COMMONBASE)/util/%.c
1512         $(COMPILE.c) -o $@ $<
1513         $(CTFCONVERT_O)
```



```
1515 $(OBJS_DIR)/%.o:                $(COMMONBASE)/unicode/%.c
1516         $(COMPILE.c) -o $@ $<
1517         $(CTFCONVERT_O)

1519 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/vm/%.c
1520         $(COMPILE.c) -o $@ $<
1521         $(CTFCONVERT_O)

1523 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/zmod/%.c
1524         $(COMPILE.c) -o $@ $<
1525         $(CTFCONVERT_O)

1527 $(OBJS_DIR)/zlib_obj.o:         $(ZLIB_OBJS:%=$(OBJS_DIR)/%)
1528         $(LD) -r -Breduce -M$(UTSBASE)/common/zmod/mapfile -o $@ \
1529             $(ZLIB_OBJS:%=$(OBJS_DIR)/%)
1530         $(CTFMERGE) -t -f -L VERSION -o $@ $(ZLIB_OBJS:%=$(OBJS_DIR)/%)

1532 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/hxge/%.c
1533         $(COMPILE.c) -o $@ $<
1534         $(CTFCONVERT_O)

1536 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/tpm/%.c
1537         $(COMPILE.c) -o $@ $<
1538         $(CTFCONVERT_O)

1540 $(OBJS_DIR)/%.o:                $(UTSBASE)/common/io/tpm/%.s
1541         $(COMPILE.s) -o $@ $<

1543 $(OBJS_DIR)/bz2%.o:             $(COMMONBASE)/bzip2/%.c
1544         $(COMPILE.c) -o $@ -I$(COMMONBASE)/bzip2 $<
1545         $(CTFCONVERT_O)

1547 BZ2LINT = -erroff=%all -I$(UTSBASE)/common/bzip2

1549 $(LINTS_DIR)/bz2%.ln:           $(COMMONBASE)/bzip2/%.c
1550          @($(LHEAD) $(LINT.c) -C $(LINTS_DIR)/‘basename $@ .ln‘ $(BZ2LINT) $< $(

1552 #
1553 # SVM
1554 #

1556 MD_XDR_CSRC     =       $(UTSBASE)/common/io/lvm/md
1557 MD_XDR_XSRC     =       $(UTSBASE)/common/sys/lvm
1558 RPCGENFLAGS     +=      -C -M -D_KERNEL -DSYSV

1560 $(MD_XDR_CSRC)/meta_basic_xdr.c:        $(MD_XDR_XSRC)/meta_basic.x
1561         $(RPCGEN) $(RPCGENFLAGS) -c -i 100 $(MD_XDR_XSRC)/meta_basic.x | \
1562         nawk ’{sub(/^#include "(\.\.\/\.\.\/)/,"#include \"\.\.\/\.\.\/\.\.\/\.\
1563         nawk ’{sub(/meta_basic.h/, "md_basic.h"); print $$0}’ >$@

1565 $(MD_XDR_CSRC)/metamed_xdr.c:   $(MD_XDR_XSRC)/metamed.x
1566         $(RPCGEN) $(RPCGENFLAGS) -c -i 100 $(MD_XDR_XSRC)/metamed.x |  \
1567         nawk ’{sub(/^#include "(\.\.\/\.\.\/)/,"#include \"\.\.\/\.\.\/\.\.\/\.\
1568         nawk ’{sub(/metamed.h/, "mdmed.h"); print $$0}’ >$@

1570 #
1571 #       Section 1b:     Lint ‘objects’
1572 #
1573 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/aes/%.c
1574         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1576 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/arcfour/%.c
1577         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1579 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/blowfish/%.c
```

```
1580         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1582 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/ecc/%.c
1583         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1585 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/modes/%.c
1586         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1588 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/padding/%.c
1589         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1591 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/rng/%.c
1592         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1594 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/rsa/%.c
1595         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1597 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/bignum/%.c
1598         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1600 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/bignum/%.c
1601         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1603 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/mpi/%.c
1604         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1606 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/acl/%.c
1607         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1609 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/avl/%.c
1610         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1612 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/ucode/%.c
1613         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1615 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/brand/sn1/%.c
1616         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1618 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/brand/solaris10/%.c
1619         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1621 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/c2/%.c
1622         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1624 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/conf/%.c
1625         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1627 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/contract/%.c
1628         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1630 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/cpr/%.c
1631         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1633 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ctf/%.c
1634         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1636 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/ctf/%.c
1637         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1639 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/pci/%.c
1640         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1642 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/devid/%.c
1643         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1645 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/des/%.c
```



```
1646         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1648 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/smbios/%.c
1649         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1651 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ncall/%.c
1652         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1654 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ns/dsw/%.c
1655         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1657 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ns/nsctl/%.c
1658         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1660 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ns/rdc/%.c
1661         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1663 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ns/sdbc/%.c
1664         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1666 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ns/solaris/%.c
1667         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1669 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ns/sv/%.c
1670         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1672 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/avs/ns/unistat/%.c
1673         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1675 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/des/%.c
1676         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1678 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/crypto/api/%.c
1679         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1681 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/crypto/core/%.c
1682         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1684 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/crypto/io/%.c
1685         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1687 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/crypto/spi/%.c
1688         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1690 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/disp/%.c
1691         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1693 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/dtrace/%.c
1694         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1696 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/exacct/%.c
1697         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1699 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/exec/aout/%.c
1700         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1702 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/exec/elf/%.c
1703         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1705 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/exec/intp/%.c
1706         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1708 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/exec/shbin/%.c
1709         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1711 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/exec/java/%.c
```

```
1712         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1714 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/%.c
1715         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1717 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/autofs/%.c
1718         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1720 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/cachefs/%.c
1721         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1723 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/ctfs/%.c
1724         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1726 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/doorfs/%.c
1727         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1729 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/dcfs/%.c
1730         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1732 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/devfs/%.c
1733         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1735 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/dev/%.c
1736         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1738 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/fd/%.c
1739         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1741 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/fifofs/%.c
1742         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1744 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/hsfs/%.c
1745         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1747 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/lofs/%.c
1748         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1750 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/mntfs/%.c
1751         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1753 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/namefs/%.c
1754         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1756 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/smbsrv/%.c
1757         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1759 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/smbsrv/%.c
1760         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1762 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/nfs/%.c
1763         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1765 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/objfs/%.c
1766         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1768 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/pcfs/%.c
1769         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1771 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/portfs/%.c
1772         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1774 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/proc/%.c
1775         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1777 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/sharefs/%.c
```



```
1778         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1780 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/smbclnt/%.c
1781         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1783 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/smbclnt/netsmb/%.c
1784         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1786 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/smbclnt/smbfs/%.c
1787         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1789 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/sockfs/%.c
1790         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1792 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/specfs/%.c
1793         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1795 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/swapfs/%.c
1796         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1798 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/tmpfs/%.c
1799         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1801 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/udfs/%.c
1802         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1804 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/ufs/%.c
1805         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1807 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/ufs_log/%.c
1808         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1810 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/vscan/%.c
1811         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1813 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/zfs/%.c
1814         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1816 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/fs/zut/%.c
1817         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1819 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/xattr/%.c
1820         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1822 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/zfs/%.c
1823         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1825 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/gssapi/%.c
1826         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1828 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/gssapi/mechs/dummy/%.c
1829         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1831 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/%.c
1832         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1834 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/%.c
1835         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1837 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/des/%.c
1838         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1840 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/dk/%.c
1841         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1843 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/os/%.c
```

```
1844         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1846 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/arcfour/%.c
1847         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1849 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/enc_provider/%.c
1850         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1852 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/hash_provider/%.c
1853         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1855 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/keyhash_provider/%.c
1856         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1858 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/raw/%.c
1859         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1861 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/crypto/old/%.c
1862         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1864 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/krb5/krb/%.c
1865         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1867 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/krb5/os/%.c
1868         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1870 $(LINTS_DIR)/%.ln:              $(KMECHKRB5_BASE)/mech/%.c
1871         @($(LHEAD) $(LINT.c) $(KGSSDFLAGS) $< $(LTAIL))

1873 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/idmap/%.c
1874         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1876 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/%.c
1877         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1879 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/sockmods/%.c
1880         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1882 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/arp/%.c
1883         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1885 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/dccp/%.c
1886         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1888 #endif /* ! codereview */
1889 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/ip/%.c
1890         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1892 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/ipnet/%.c
1893         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1895 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/iptun/%.c
1896         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1898 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/ipf/%.c
1899         @($(LHEAD) $(LINT.c) $(IPFFLAGS) $< $(LTAIL))

1901 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/kssl/%.c
1902         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1904 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/net/patricia/%.c
1905         @($(LHEAD) $(LINT.c) $(IPFFLAGS) $< $(LTAIL))

1907 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/udp/%.c
1908         @($(LHEAD) $(LINT.c) $< $(LTAIL))
```



```
1910 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/sctp/%.c
1911         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1913 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/tcp/%.c
1914         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1916 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/ilb/%.c
1917         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1919 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/nca/%.c
1920         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1922 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/dlpistub/%.c
1923         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1925 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/%.c
1926         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1928 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/1394/%.c
1929         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1931 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/1394/adapters/%.c
1932         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1934 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/1394/targets/av1394/%.c
1935         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1937 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/1394/targets/dcam1394/%.c
1938         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1940 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/1394/targets/scsa1394/%.c
1941         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1943 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sbp2/%.c
1944         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1946 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/aac/%.c
1947         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1949 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/afe/%.c
1950         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1952 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/atge/%.c
1953         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1955 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/arn/%.c
1956         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1958 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ath/%.c
1959         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1961 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/atu/%.c
1962         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1964 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/impl/%.c
1965         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1967 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/ac97/%.c
1968         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1970 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audio1575/%.c
1971         @($(LHEAD) $(LINT.c) $< $(LTAIL))

1973 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audio810/%.c
1974         @($(LHEAD) $(LINT.c) $< $(LTAIL))
```

```
$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiocmi/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiocmihd/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audioens/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audioemu10k/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiohd/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audioixp/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiols/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiopci/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiop16x/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiosolo/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiots/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiovia823x/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/audio/drv/audiovia97/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/bfe/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/bpf/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/bge/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/blkdev/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/cardbus/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/lu/stmf_sbd/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/port/fct/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/port/qlt/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/port/srpt/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))
```



```
$(LINTS_DIR)/%.ln:              $(COMMONBASE)/iscsit/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/port/fcoet/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/port/iscsit/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/port/pppt/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/comstar/stmf/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/dld/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/dls/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/dmfe/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/drm/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/efe/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/elxl/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fcoe/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/hme/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/pciex/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/hotplug/hpcsvc/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/pciex/hotplug/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/hotplug/pcihp/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/rds/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/rdsv3/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/iser/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/ibd/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/eoib/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))
```

```
2108 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/of/sol_ofs/%.c
2109         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2111 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/of/sol_ucma/%.c
2112         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2114 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/of/sol_umad/%.c
2115         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2117 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/of/sol_uverbs/%.
2118         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2120 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/sdp/%.c
2121         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2123 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/mgt/ibcm/%.c
2124         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2126 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/mgt/ibdm/%.c
2127         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2129 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/mgt/ibdma/%.c
2130         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2132 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/mgt/ibmf/%.c
2133         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2135 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/ibnex/%.c
2136         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2138 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/ibtl/%.c
2139         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2141 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/adapters/tavor/%.c
2142         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2144 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/adapters/hermon/%.c
2145         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2147 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ib/clients/daplt/%.c
2148         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2150 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/iscsi/%.c
2151         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2153 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/idm/%.c
2154         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2156 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ipw/%.c
2157         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2159 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/iwh/%.c
2160         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2162 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/iwi/%.c
2163         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2165 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/iwk/%.c
2166         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2168 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/iwp/%.c
2169         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2171 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/kb8042/%.c
2172         @($(LHEAD) $(LINT.c) $< $(LTAIL))
```



```
2174 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/kbtrans/%.c
2175         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2177 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ksocket/%.c
2178         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2180 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/aggr/%.c
2181         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2183 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lp/%.c
2184         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2186 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/hotspares/%.c
2187         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2189 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/md/%.c
2190         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2192 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/mirror/%.c
2193         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2195 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/raid/%.c
2196         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2198 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/softpart/%.c
2199         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2201 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/stripe/%.c
2202         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2204 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/notify/%.c
2205         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2207 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/lvm/trans/%.c
2208         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2210 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mac/%.c
2211         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2213 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mac/plugins/%.c
2214         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2216 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mega_sas/%.c
2217         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2219 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mii/%.c
2220         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2222 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mr_sas/%.c
2223         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2225 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/adapters/mpt_sas/%.c
2226         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2228 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mxfe/%.c
2229         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2231 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mwl/%.c
2232         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2234 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/mwl/mwl_fw/%.c
2235         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2237 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/net80211/%.c
2238         @($(LHEAD) $(LINT.c) $< $(LTAIL))
```

```
$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/nge/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/nxge/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/nxge/%.s
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/nxge/npi/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/pci-ide/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/pcmcia/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/pcan/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/pcn/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/pcwl/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ppp/sppp/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ppp/spppasyn/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ppp/spppptun/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ral/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/rge/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/rtls/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/rsm/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/rtw/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/rum/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/rwd/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/rwn/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sata/adapters/ahci/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sata/adapters/nv_sata/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))
```



```
$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sata/adapters/si3124/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sata/impl/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/adapters/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/adapters/blk2scsa/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/adapters/pmcs/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/fop
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fibre-channel/ulp/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fibre-channel/impl/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fibre-channel/fca/qlc/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fibre-channel/fca/qlge/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fibre-channel/fca/emlxs/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fibre-channel/fca/oce/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/fibre-channel/fca/fcoei/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/conf/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/impl/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/targets/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sdcard/adapters/sdhost/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sdcard/impl/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sdcard/targets/sdcard/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/sfe/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/simnet/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))
```

```
2372 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/softmac/%.c
2373         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2375 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/uath/%.c
2376         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2378 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/uath/uath_fw/%.c
2379         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2381 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ural/%.c
2382         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2384 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/urtw/%.c
2385         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2387 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/audio/usb_ac/%.
2388         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2390 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/audio/usb_as/%.
2391         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2393 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/audio/usb_ah/%.
2394         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2396 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbskel/%.c
2397         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2399 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/video/usbvc/%.c
2400         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2402 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/hwarc/%.c
2403         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2405 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/hid/%.c
2406         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2408 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/hidparser/%.c
2409         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2411 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbkbm/%.c
2412         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2414 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbms/%.c
2415         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2417 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbinput/usbwcm
2418         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2420 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/ugen/%.c
2421         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2423 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/printer/%.c
2424         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2426 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbser/%.c
2427         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2429 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbser/usbsacm/
2430         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2432 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbser/usbftdi/
2433         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2435 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbser/usbser_k
2436         @($(LHEAD) $(LINT.c) $< $(LTAIL))
```



```
2438 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbser/usbsprl/
2439         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2441 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/wusb_df/%.c
2442         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2444 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/hwa1480_fw/%.c
2445         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2447 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/wusb_ca/%.c
2448         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2450 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/clients/usbecm/%.c
2451         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2453 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/hcd/openhci/%.c
2454         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2456 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/hcd/ehci/%.c
2457         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2459 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/hcd/uhci/%.c
2460         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2462 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/hubd/%.c
2463         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2465 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/scsa2usb/%.c
2466         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2468 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/usb_mid/%.c
2469         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2471 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/usb_ia/%.c
2472         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2474 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/usba/%.c
2475         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2477 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/usba10/%.c
2478         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2480 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/uwb/uwba/%.c
2481         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2483 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/usb/hwa/hwahc/%.c
2484         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2486 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/vuidmice/%.c
2487         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2489 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/vnic/%.c
2490         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2492 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/wpi/%.c
2493         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2495 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/zyd/%.c
2496         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2498 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/chxge/com/%.c
2499         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2501 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/chxge/%.c
2502         @($(LHEAD) $(LINT.c) $< $(LTAIL))
```

```
$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ixgb/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/xge/drv/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/xge/hal/xgehal/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/e1000g/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/igb/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/iprb/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ixgbe/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/ntxn/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/myri10ge/drv/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ipp/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ipp/ipgpc/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ipp/dlcosmk/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ipp/flowacct/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ipp/dscpmk/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ipp/meters/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/kiconv/kiconv_emea/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/kiconv/kiconv_ja/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/kiconv/kiconv_ko/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/kiconv/kiconv_sc/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/kiconv/kiconv_tc/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/kmdb/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/krtld/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))
```



```
$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/ktli/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/list/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/lvm/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/lzma/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/md4/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/md5/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/net/dhcp/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/nvpair/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/os/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/rpc/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/pcmcia/cs/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/pcmcia/cis/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/pcmcia/nexus/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/pcmcia/pcs/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/rpc/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/rpc/sec/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/rpc/sec_gss/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/sha1/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/crypto/sha2/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/syscall/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(UTSBASE)/common/tnf/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))

$(LINTS_DIR)/%.ln:              $(COMMONBASE)/tsol/%.c
        @($(LHEAD) $(LINT.c) $< $(LTAIL))
```

```
2636 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/util/%.c
2637         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2639 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/unicode/%.c
2640         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2642 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/vm/%.c
2643         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2645 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/scsi/adapters/iscsi/%.c
2646         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2648 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/iscsi/%.c
2649         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2651 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/inet/kifconf/%.c
2652         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2654 ZMODLINTFLAGS = -erroff=E_CONSTANT_CONDITION

2656 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/zmod/%.c
2657         @($(LHEAD) $(LINT.c) $(ZMODLINTFLAGS) $< $(LTAIL))

2659 $(LINTS_DIR)/zlib_obj.ln:       $(ZLIB_OBJS:%.o=$(LINTS_DIR)/%.ln) \
2660                                 $(UTSBASE)/common/zmod/zlib_lint.c
2661         @($(LHEAD) $(LINT.c) -C $(LINTS_DIR)/zlib_obj \
2662             $(UTSBASE)/common/zmod/zlib_lint.c $(LTAIL))

2664 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/hxge/%.c
2665         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2667 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/tpm/%.c
2668         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2670 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/tpm/%.s
2671         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2673 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/vr/%.c
2674         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2676 $(LINTS_DIR)/%.ln:              $(UTSBASE)/common/io/yge/%.c
2677         @($(LHEAD) $(LINT.c) $< $(LTAIL))

2679 $(LINTS_DIR)/%.ln:              $(COMMONBASE)/fsreparse/%.c
2680         @($(LHEAD) $(LINT.c) $< $(LTAIL))
```

```
**********************************************************
   25677 Mon Jul  9 14:38:12 2012
new/usr/src/uts/common/inet/dccp/dccp.c
dccp: starting module template
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * This file implements the Data Congestion Control Protocol (DCCP).
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#include <sys/strsun.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/sockio.h>
#include <sys/vtrace.h>
#include <sys/sdt.h>
#include <sys/debug.h>
#include <sys/ddi.h>
#include <sys/isa_defs.h>
#include <sys/policy.h>
#include <sys/tsol/label.h>
#include <sys/tsol/tnet.h>
#include <inet/kstatcom.h>
#include <inet/snmpcom.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"
#include "dccp_stack.h"

int dccp_squeue_flag;

/* Setable in /etc/system */
uint_t dccp_bind_fanout_size = DCCP_BIND_FANOUT_SIZE;
```



```
static void     dccp_notify(void *, ip_xmit_attr_t *, ixa_notify_type_t,
                    ixa_notify_arg_t);

/* Functions to register netstack */
static void     *dccp_stack_init(netstackid_t, netstack_t *);
static void     dccp_stack_fini(netstackid_t, void *);

/* Stream device open functions */
static int      dccp_openv4(queue_t *, dev_t *, int, int, cred_t *);
static int      dccp_openv6(queue_t *, dev_t *, int, int, cred_t *);
static int      dccp_open(queue_t *, dev_t *, int, int, cred_t *,
                    boolean_t);

/* Write service routine */
static void     dccp_wsrv(queue_t *);

/* Connection related functions */
static int      dccp_connect_ipv4(dccp_t *, ipaddr_t *, in_port_t, uint_t);
static int      dccp_connect_ipv6(dccp_t *, in6_addr_t *, in_port_t, uint32_t,
    uint_t, uint32_t);

/* Initialise ISS */
static void     dccp_iss_init(dccp_t *);

struct module_info dccp_rinfo = {
        DCCP_MOD_ID, DCCP_MOD_NAME, 0, INFPSZ, DCCP_RECV_HIWATER,
        DCCP_RECV_LOWATER
};

static struct module_info dccp_winfo = {
        DCCP_MOD_ID, DCCP_MOD_NAME, 0, INFPSZ, 127, 16
};

/*
 * Queue information structure with DCCP entry points.
 */
struct qinit dccp_rinitv4 = {
        NULL, (pfi_t)dccp_rsrv, dccp_openv4, dccp_tpi_close, NULL, &dccp_rinfo
};

struct qinit dccp_rinitv6 = {
        NULL, (pfi_t)dccp_rsrv, dccp_openv6, dccp_tpi_close, NULL, &dccp_rinfo
};

struct qinit dccp_winit = {
        (pfi_t)dccp_wput, (pfi_t)dccp_wsrv, NULL, NULL, NULL, &dccp_winfo
};

/* Initial entry point for TCP in socket mode */
struct qinit dccp_sock_winit = {
        (pfi_t)dccp_wput_sock, (pfi_t)dccp_wsrv, NULL, NULL, NULL, &dccp_winfo
};

struct qinit dccp_fallback_sock_winit = {
        (pfi_t)dccp_wput_fallback, NULL, NULL, NULL, NULL, &dccp_winfo
};
/*
 * DCCP as acceptor STREAM.
 */
struct qinit dccp_acceptor_rinit = {
        NULL, (pfi_t)dccp_rsrv, NULL, dccp_tpi_close_accept, NULL, &dccp_winfo
};

struct qinit dccp_acceptor_winit = {
        (pfi_t)dccp_tpi_accept, NULL, NULL, NULL, NULL, &dccp_winfo
};
```

```
129 /* AF_INET /dev/dccp */
130 struct streamtab dccpinfov4 = {
131         &dccp_rinitv4, &dccp_winit
132 };

134 /* AF_INET6 /dev/dccp6 */
135 struct streamtab dccpinfov6 = {
136         &dccp_rinitv6, &dccp_winit
137 };

139 /* Template for response to info request */
140 struct T_info_ack dccp_g_t_info_ack = {
141         T_INFO_ACK,             /* PRIM_type */
142         0,                      /* TSDU_size */
143         T_INFINITE,             /* ETSDU_size */
144         T_INVALID,              /* CDATA_size */
145         T_INVALID,              /* DDATA_size */
146         sizeof (sin_t),         /* ADDR_size */
147         0,                      /* OPT_size - not initialized here */
148         TIDUSZ,                 /* TIDU_size */
149         T_COTS_ORD,             /* SERV_type */
150         DCCPS_IDLE,             /* CURRENT_state */
151         (XPG4_1|EXPINLINE)      /* PROVIDER_flag */
152 };

154 struct T_info_ack dccp_g_t_info_ack_v6 = {
155         T_INFO_ACK,             /* PRIM_type */
156         0,                      /* TSDU_size */
157         T_INFINITE,             /* ETSDU_size */
158         T_INVALID,              /* CDATA_size */
159         T_INVALID,              /* DDATA_size */
160         sizeof (sin6_t),        /* ADDR_size */
161         0,                      /* OPT_size - not initialized here */
162         TIDUSZ,                 /* TIDU_size */
163         T_COTS_ORD,             /* SERV_type */
164         DCCPS_IDLE,             /* CURRENT_state */
165         (XPG4_1|EXPINLINE)      /* PROVIDER_flag */
166 };

168 /*
169  * DCCP Tunables.
170  */
171 extern mod_prop_info_t dccp_propinfo_tbl[];
172 extern int dccp_propinfo_count;

174 /*
175  * Register DCCP in ip netstack.
176  */
177 void
178 dccp_ddi_g_init(void)
179 {
180         netstack_register(NS_DCCP, dccp_stack_init, NULL, dccp_stack_fini);
181 }

183 /*
184  * Unregister DCCP from ip netstack.
185  */
186 void
187 dccp_ddi_g_destroy(void)
188 {
189         netstack_unregister(NS_DCCP);
190 }

192 #define INET_NAME       "ip"
```



```
194 /*
195  * Initialize this DCCP stack instance.
196  */
197 static void *
198 dccp_stack_init(netstackid_t stackid, netstack_t *ns)
199 {
200         dccp_stack_t    *dccps;
201         major_t         major;
202         size_t          arrsz;
203         int             error;
204         int             i;

206         dccps = kmem_zalloc(sizeof (*dccps), KM_SLEEP);
207         if (dccps == NULL) {
208                 return (NULL);
209         }
210         dccps->dccps_netstack = ns;

212         /* Ports */
213         mutex_init(&dccps->dccps_epriv_port_lock, NULL, MUTEX_DEFAULT, NULL);
214         dccps->dccps_num_epriv_ports = DCCP_NUM_EPRIV_PORTS;
215         dccps->dccps_epriv_ports[0] = ULP_DEF_EPRIV_PORT1;
216         dccps->dccps_epriv_ports[1] = ULP_DEF_EPRIV_PORT2;
217         dccps->dccps_min_anonpriv_port = 512;

219         dccps->dccps_bind_fanout_size = dccp_bind_fanout_size;

221         /* Bind fanout */
222         dccps->dccps_bind_fanout = kmem_zalloc(dccps->dccps_bind_fanout_size *
223             sizeof (dccp_df_t), KM_SLEEP);
224         for (i = 0; i < dccps->dccps_bind_fanout_size; i++) {
225                 mutex_init(&dccps->dccps_bind_fanout[i].df_lock, NULL,
226                     MUTEX_DEFAULT, NULL);
227         }

229         /* Tunable properties */
230         arrsz = dccp_propinfo_count * sizeof (mod_prop_info_t);
231         dccps->dccps_propinfo_tbl = kmem_alloc(arrsz, KM_SLEEP);
232         if (dccps->dccps_propinfo_tbl == NULL) {
233                 kmem_free(dccps, sizeof (*dccps));
234                 return (NULL);
235         }
236         bcopy(dccp_propinfo_tbl, dccps->dccps_propinfo_tbl, arrsz);

238         /* Allocate per netstack cpu stats */
239         mutex_enter(&cpu_lock);
240         dccps->dccps_sc_cnt = MAX(ncpus, boot_ncpus);
241         mutex_exit(&cpu_lock);

243         dccps->dccps_sc = kmem_zalloc(max_ncpus * sizeof (dccp_stats_cpu_t *),
244             KM_SLEEP);
245         for (i = 0; i < dccps->dccps_sc_cnt; i++) {
246                 dccps->dccps_sc[i] = kmem_zalloc(sizeof (dccp_stats_cpu_t),
247                     KM_SLEEP);
248         }

250         /* Driver major number */
251         major = mod_name_to_major(INET_NAME);
252         error = ldi_ident_from_major(major, &dccps->dccps_ldi_ident);
253         ASSERT(error == 0);

255         return (dccps);
256 }

258 /*
259  * Destroy this DCCP netstack instance.
```

```
 */
static void
dccp_stack_fini(netstackid_t stackid, void *arg)
{
        dccp_stack_t    *dccps = (dccp_stack_t *)arg;
        int             i;

        /* Cpu stats */
        for (i = 0; i < dccps->dccps_sc_cnt; i++) {
                kmem_free(dccps->dccps_sc[i], sizeof (dccp_stats_cpu_t));
        }
        kmem_free(dccps->dccps_sc, max_ncpus * sizeof (dccp_stats_cpu_t *));

        /* Tunable properties */
        kmem_free(dccps->dccps_propinfo_tbl,
            dccp_propinfo_count * sizeof (mod_prop_info_t));
        dccps->dccps_propinfo_tbl = NULL;

        /* Bind fanout */
        for (i = 0; i < dccps->dccps_bind_fanout_size; i++) {
                ASSERT(dccps->dccps_bind_fanout[i].df_dccp == NULL);
                mutex_destroy(&dccps->dccps_bind_fanout[i].df_lock);
        }
        kmem_free(dccps->dccps_bind_fanout, dccps->dccps_bind_fanout_size *
            sizeof (dccp_df_t));
        dccps->dccps_bind_fanout = NULL;

        kmem_free(dccps, sizeof (*dccps));
}

/* /dev/dccp */
static int
dccp_openv4(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
{
        cmn_err(CE_NOTE, "dccp.c: dccp_openv4\n");

        return (dccp_open(q, devp, flag, sflag, credp, B_FALSE));
}

/* /dev/dccp6 */
static int
dccp_openv6(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
{
        cmn_err(CE_NOTE, "dccp.c: dccp_openv6\n");

        return (dccp_open(q, devp, flag, sflag, credp, B_TRUE));
}

/*
 * Common open function for v4 and v6 devices.
 */
static int
dccp_open(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp,
    boolean_t isv6)
{
        conn_t          *connp;
        dccp_t          *dccp;
        vmem_t          *minor_arena;
        dev_t           conn_dev;
        boolean_t       issocket;
        int             error;

        cmn_err(CE_NOTE, "dccp.c: dccp_open");

        /* If the stream is already open, return immediately */
        if (q->q_ptr != NULL) {
```



```
                return (0);
        }

        if (sflag == MODOPEN) {
                return (EINVAL);
        }

        if ((ip_minor_arena_la != NULL) && (flag & SO_SOCKSTR) &&
            ((conn_dev = inet_minor_alloc(ip_minor_arena_la)) != 0)) {
                minor_arena = ip_minor_arena_la;
        } else {
                /*
                 * Either minor numbers in the large arena were exhausted
                 * or a non socket application is doing the open.
                 * Try to allocate from the small arena.
                */
                if ((conn_dev = inet_minor_alloc(ip_minor_arena_sa)) == 0) {
                        return (EBUSY);
                }
                minor_arena = ip_minor_arena_sa;
        }

        ASSERT(minor_arena != NULL);

        *devp = makedevice(getmajor(*devp), (minor_t)conn_dev);

        if (flag & SO_FALLBACK) {
                /*
                 * Non streams socket needs a stream to fallback to.
                 */
                RD(q)->q_ptr = (void *)conn_dev;
                WR(q)->q_qinfo = &dccp_fallback_sock_winit;
                WR(q)->q_ptr = (void *)minor_arena;
                qprocson(q);
                return (0);
        } else if (flag & SO_ACCEPTOR) {
                q->q_qinfo = &dccp_acceptor_rinit;
                /*
                 * The conn_dev and minor_arena will be subsequently used by
                 * dccp_tli_accept() and dccp_tpi_close_accept() to figure out
                 * the minor device number for this connection from the q_ptr.
                 */
                RD(q)->q_ptr = (void *)conn_dev;
                WR(q)->q_qinfo = &dccp_acceptor_winit;
                WR(q)->q_ptr = (void *)minor_arena;
                qprocson(q);
                return (0);
        }


        issocket = flag & SO_SOCKSTR;
        connp = dccp_create_common(credp, isv6, issocket, &error);
        if (connp == NULL) {
                inet_minor_free(minor_arena, conn_dev);
                q->q_ptr = WR(q)->q_ptr = NULL;
                return (error);
        }

        connp->conn_rq = q;
        connp->conn_wq = WR(q);
        q->q_ptr = WR(q)->q_ptr = connp;

        connp->conn_dev = conn_dev;
        connp->conn_minor_arena = minor_arena;

        ASSERT(q->q_qinfo == &dccp_rinitv4 || q->q_qinfo == &dccp_rinitv6);
```

```
392         ASSERT(WR(q)->q_qinfo == &dccp_winit);

394         dccp = connp->conn_dccp;

396         if (issocket) {
397                 WR(q)->q_qinfo = &dccp_sock_winit;
398         } else {
399 #ifdef  _ILP32
400                 dccp->dccp_acceptor_id = (t_uscalar_t)RD(q);
401 #else
402                 dccp->dccp_acceptor_id = conn_dev;
403 #endif  /* _ILP32 */
404         }

406         /*
407          * Put the ref for DCCP. Ref for IP was already put
408          * by ipcl_conn_create. Also Make the conn_t globally
409          * visible to walkers.
410          */
411         mutex_enter(&connp->conn_lock);
412         CONN_INC_REF_LOCKED(connp);
413         ASSERT(connp->conn_ref == 2);
414         connp->conn_state_flags &= ~CONN_INCIPIENT;
415         mutex_exit(&connp->conn_lock);

417         qprocson(q);

419         return (0);
420 }

422 /*
423  * IXA notify
424  */
425 static void
426 dccp_notify(void *arg, ip_xmit_attr_t *ixa, ixa_notify_type_t ntype,
427     ixa_notify_arg_t narg)
428 {
429         cmn_err(CE_NOTE, "dccp.c: dccp_notify");
430 }

432 /*
433  * Build the template headers.
434  */
435 int
436 dccp_build_hdrs(dccp_t *dccp)
437 {
438         dccp_stack_t    *dccps = dccp->dccp_dccps;
439         conn_t          *connp = dccp->dccp_connp;
440         dccpha_t        *dccpha;
441         uint32_t        cksum;
442         char            buf[DCCP_MAX_HDR_LENGTH];
443         uint_t          buflen;
444         uint_t          ulplen = 12;
445         uint_t          extralen = 0;
446         int             error;

448         cmn_err(CE_NOTE, "dccp.c: dccp_build_hdrs");

450         buflen = connp->conn_ht_ulp_len;
451         if (buflen != 0) {
452                 cmn_err(CE_NOTE, "buflen != 0");
453                 bcopy(connp->conn_ht_ulp, buf, buflen);
454                 extralen -= buflen - ulplen;
455                 ulplen = buflen;
456         }
```



```
458         mutex_enter(&connp->conn_lock);
459         error = conn_build_hdr_template(connp, ulplen, extralen,
460             &connp->conn_laddr_v6, &connp->conn_faddr_v6, connp->conn_flowinfo);
461         mutex_exit(&connp->conn_lock);
462         if (error != 0) {
463                 cmn_err(CE_NOTE, "conn_build_hdr_template failed");
464                 return (error);
465         }

467         dccpha = (dccpha_t *)connp->conn_ht_ulp;
468         dccp->dccp_dccpha = dccpha;

470         if (buflen != 0) {
471                 bcopy(buf, connp->conn_ht_ulp, buflen);
472         } else {
473                 dccpha->dha_sum = 0;
474                 dccpha->dha_lport = connp->conn_lport;
475                 dccpha->dha_fport = connp->conn_fport;
476         }

478         cksum = sizeof (dccpha_t) + connp->conn_sum;
479         cksum = (cksum >> 16) + (cksum & 0xFFFF);
480         dccpha->dha_sum = htons(cksum);
481         dccpha->dha_offset = 7;
482         dccpha->dha_x = 1;

484         if (connp->conn_ipversion == IPV4_VERSION) {
485                 dccp->dccp_ipha = (ipha_t *)connp->conn_ht_iphc;
486         } else {
487                 dccp->dccp_ip6h = (ip6_t *)connp->conn_ht_iphc;
488         }

490         /* XXX */

492         return (0);
493 }

495 /*
496  * DCCP write service routine.
497  */
498 static void
499 dccp_wsrv(queue_t *q)
500 {
501         /* XXX:DCCP */
502 }

504 /*
505  * Common create function for streams and sockets.
506  */
507 conn_t *
508 dccp_create_common(cred_t *credp, boolean_t isv6, boolean_t issocket,
509     int *errorp)
510 {
511         conn_t          *connp;
512         dccp_t          *dccp;
513         dccp_stack_t    *dccps;
514         netstack_t      *ns;
515         squeue_t        *sqp;
516         zoneid_t        zoneid;

518         cmn_err(CE_NOTE, "dccp.c: dccp_create_common\n");

520         ASSERT(errorp != NULL);

522         *errorp = secpolicy_basic_net_access(credp);
523         if (*errorp != 0) {
```

```
524                 return (NULL);
525         }

527         /*
528          * Find the right netstack
529          */
530         ns = netstack_find_by_cred(credp);
531         ASSERT(ns != NULL);
532         dccps = ns->netstack_dccp;
533         ASSERT(dccps != NULL);

535         /*
536          * XXX
537          */
538         if (ns->netstack_stackid != GLOBAL_NETSTACKID) {
539                 zoneid = GLOBAL_ZONEID;
540         } else {
541                 zoneid = crgetzoneid(credp);
542         }

544         sqp = IP_SQUEUE_GET((uint_t)gethrtime());
545         connp = (conn_t *)dccp_get_conn(sqp, dccps);
546         netstack_rele(dccps->dccps_netstack);
547         if (connp == NULL) {
548                 *errorp = ENOSR;
549                 return (NULL);
550         }
551         ASSERT(connp->conn_ixa->ixa_protocol == connp->conn_proto);

553         connp->conn_sqp = sqp;
554         connp->conn_initial_sqp = connp->conn_sqp;
555         connp->conn_ixa->ixa_sqp = connp->conn_sqp;
556         dccp = connp->conn_dccp;

558         /* Setting flags for ip output */
559         connp->conn_ixa->ixa_flags |= IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
560             IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO;

562         ASSERT(connp->conn_proto == IPPROTO_DCCP);
563         ASSERT(connp->conn_dccp == dccp);
564         ASSERT(dccp->dccp_connp == connp);

566         if (isv6) {
567                 connp->conn_ixa->ixa_src_preferences = IPV6_PREFER_SRC_DEFAULT;
568                 connp->conn_ipversion = IPV6_VERSION;
569                 connp->conn_family = AF_INET6;
570                 /* XXX mms, ttl */
571         } else {
572                 connp->conn_ipversion = IPV4_VERSION;
573                 connp->conn_family = AF_INET;
574                 /* XXX mms, ttl */
575         }
576         connp->conn_xmit_ipp.ipp_unicast_hops = connp->conn_default_ttl;

578         crhold(credp);
579         connp->conn_cred = credp;
580         connp->conn_cpid = curproc->p_pid;
581         connp->conn_open_time = ddi_get_lbolt64();

583         ASSERT(!(connp->conn_ixa->ixa_free_flags & IXA_FREE_CRED));
584         connp->conn_ixa->ixa_cred = credp;
585         connp->conn_ixa->ixa_cpid = connp->conn_cpid;

587         connp->conn_zoneid = zoneid;
588         connp->conn_zone_is_global = (crgetzoneid(credp) == GLOBAL_ZONEID);
589         connp->conn_ixa->ixa_zoneid = zoneid;
```



```
590         connp->conn_mlp_type = mlptSingle;


593         dccp->dccp_dccps = dccps;
594         dccp->dccp_state = DCCPS_CLOSED;

596         ASSERT(connp->conn_netstack == dccps->dccps_netstack);
597         ASSERT(dccp->dccp_dccps == dccps);

599         /* XXX rcvbuf, sndbuf etc */

601         connp->conn_so_type = SOCK_STREAM;

603         SOCK_CONNID_INIT(dccp->dccp_connid);
604         dccp_init_values(dccp, NULL);

606         return (connp);
607 }

609 /*
610  * Common close function for streams and sockets.
611  */
612 void
613 dccp_close_common(conn_t *connp, int flags)
614 {
615         dccp_t          *dccp = connp->conn_dccp;
616         boolean_t       conn_ioctl_cleanup_reqd = B_FALSE;

618         ASSERT(connp->conn_ref >= 2);

620         mutex_enter(&connp->conn_lock);
621         connp->conn_state_flags |= CONN_CLOSING;
622         if (connp->conn_oper_pending_ill != NULL) {
623                 conn_ioctl_cleanup_reqd = B_TRUE;
624         }

626         CONN_INC_REF_LOCKED(connp);
627         mutex_exit(&connp->conn_lock);

629         //ipcl_conn_destroy(connp);
630 }

632 /*
633  * Common bind function.
634  */
635 int
636 dccp_do_bind(conn_t *connp, struct sockaddr *sa, socklen_t len, cred_t *cr,
637     boolean_t bind_to_req_port_only)
638 {
639         dccp_t  *dccp = connp->conn_dccp;
640         int     error;

642         cmn_err(CE_NOTE, "dccp.c: dccp_do_bind");

644         if (dccp->dccp_state >= DCCPS_BOUND) {
645                 if (connp->conn_debug) {
646                         (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
647                             "dccp_bind: bad state, %d", dccp->dccp_state);
648                 }
649                 return (-TOUTSTATE);
650         }

652         error = dccp_bind_check(connp, sa, len, cr, bind_to_req_port_only);
653         if (error != 0) {
654                 return (error);
655         }
```

```
657         ASSERT(dccp->dccp_state == DCCPS_LISTEN);
658         /* XXX dccp_conn_req_max = 0 */
660         return (0);
661 }
663 /*
664  * Common unbind function.
665  */
666 int
667 dccp_do_unbind(conn_t *connp)
668 {
669         dccp_t  *dccp = connp->conn_dccp;
671         cmn_err(CE_NOTE, "dccp.c: dccp_do_unbind");
673         switch (dccp->dccp_state) {
674         case DCCPS_BOUND:
675         case DCCPS_LISTEN:
676                 break;
677         default:
678                 return (-TOUTSTATE);
679         }
681         /* XXX:DCCP */
683         return (0);
684 }
686 /*
687  * Common listen function.
688  */
689 int
690 dccp_do_listen(conn_t *connp, struct sockaddr *sa, socklen_t len,
691     int backlog, cred_t *cr, boolean_t bind_to_req_port_only)
692 {
693         dccp_t          *dccp = connp->conn_dccp;
694         dccp_stack_t    *dccps = dccp->dccp_dccps;
695         int32_t         oldstate;
696         int             error;
698         cmn_err(CE_NOTE, "dccp.c: dccp_do_listen");
700         /* All Solaris components should pass a cred for this operation */
701         ASSERT(cr != NULL);
703         if (dccp->dccp_state >= DCCPS_BOUND) {
705                 if ((dccp->dccp_state == DCCPS_BOUND ||
706                     dccp->dccp_state == DCCPS_LISTEN) && backlog > 0) {
707                         goto do_listen;
708                 }
709                 cmn_err(CE_NOTE, "DCCPS_BOUND, bad state");
711                 if (connp->conn_debug) {
712                         (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
713                             "dccp_listen: bad state, %d", dccp->dccp_state);
714                 }
715                 return (-TOUTSTATE);
716         } else {
717                 if (sa == NULL) {
718                         sin6_t  addr;
719                         sin6_t  *sin6;
720                         sin_t   *sin;
```



```
722                         ASSERT(IPCL_IS_NONSTR(connp));
724                         if (connp->conn_family == AF_INET) {
725                                 len = sizeof (sin_t);
726                                 sin = (sin_t *)&addr;
727                                 *sin = sin_null;
728                                 sin->sin_family = AF_INET;
729                         } else {
730                                 ASSERT(connp->conn_family == AF_INET6);
732                                 len = sizeof (sin6_t);
733                                 sin6 = (sin6_t *)&addr;
734                                 *sin6 = sin6_null;
735                                 sin6->sin6_family = AF_INET6;
736                         }
738                         sa = (struct sockaddr *)&addr;
739                 }
741                 error = dccp_bind_check(connp, sa, len, cr,
742                     bind_to_req_port_only);
743                 if (error != 0) {
744                         cmn_err(CE_NOTE, "dccp_bind_check failed");
745                         return (error);
746                 }
747                 /* Fall through and do the fanout insertion */
748         }
750 do_listen:
751         ASSERT(dccp->dccp_state == DCCPS_BOUND ||
752             dccp->dccp_state == DCCPS_LISTEN);
754         /* XXX backlog */
756         connp->conn_recv = dccp_input_listener_unbound;
758         /* Insert into the classifier table */
759         error = ip_laddr_fanout_insert(connp);
760         if (error != 0) {
761                 /* Error - undo the bind */
762                 oldstate = dccp->dccp_state;
763                 dccp->dccp_state = DCCPS_CLOSED;
765                 connp->conn_bound_addr_v6 = ipv6_all_zeros;
767                 connp->conn_laddr_v6 = ipv6_all_zeros;
768                 connp->conn_saddr_v6 = ipv6_all_zeros;
769                 connp->conn_ports = 0;
771                 if (connp->conn_anon_port) {
772                         zone_t  *zone;
774                         zone = crgetzone(cr);
775                         connp->conn_anon_port = B_FALSE;
776                         (void) tsol_mlp_anon(zone, connp->conn_mlp_type,
777                             connp->conn_proto, connp->conn_lport, B_FALSE);
778                 }
779                 connp->conn_mlp_type = mlptSingle;
781                 /* XXX dccp_bind_hash_remove */
783                 return (error);
784         } else {
785                 /* XXX connection limits */
786         }
```

```
788         return (error);
789 }

791 /*
792  * Common connect function.
793  */
794 int
795 dccp_do_connect(conn_t *connp, const struct sockaddr *sa, socklen_t len,
796     cred_t *cr, pid_t pid)
797 {
798         dccp_t          *dccp = connp->conn_dccp;
799         dccp_stack_t    *dccps = dccp->dccp_dccps;
800         ip_xmit_attr_t  *ixa = connp->conn_ixa;
801         mblk_t          *req_mp;
802         sin_t           *sin = (sin_t *)sa;
803         sin6_t          *sin6 = (sin6_t *)sa;
804         ipaddr_t        *dstaddrp;
805         in_port_t       dstport;
806         uint_t          srcid;
807         int32_t         oldstate;
808         int             error;

810         cmn_err(CE_NOTE, "dccp.c: dccp_do_connect");

812         oldstate = dccp->dccp_state;

814         switch (len) {
815         case sizeof (sin_t):
816                 sin = (sin_t *)sa;
817                 if (sin->sin_port == 0) {
818                         return (-TBADADDR);
819                 }
820                 if (connp->conn_ipv6_v6only) {
821                         return (EAFNOSUPPORT);
822                 }
823                 break;

825         case sizeof (sin6_t):
826                 sin6 = (sin6_t *)sa;
827                 if (sin6->sin6_port == 0) {
828                         return (-TBADADDR);
829                 }
830                 break;

832         default:
833                 return (EINVAL);
834         }

836         if (connp->conn_family == AF_INET6 &&
837             connp->conn_ipversion == IPV6_VERSION &&
838             IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
839                 if (connp->conn_ipv6_v6only) {
840                         return (EADDRNOTAVAIL);
841                 }

843                 connp->conn_ipversion = IPV4_VERSION;
844         }

846         switch (dccp->dccp_state) {
847         case DCCPS_LISTEN:
848                 if (IPCL_IS_NONSTR(connp)) {
849                         return (EOPNOTSUPP);
850                 }

852         case DCCPS_CLOSED:
853                 /* XXX */
```



```
854                 break;

856         default:
857                 return (-TOUTSTATE);
858         }

860         if (connp->conn_cred != cr) {
861                 crhold(cr);
862                 crfree(connp->conn_cred);
863                 connp->conn_cred = cr;
864         }
865         connp->conn_cpid = pid;

867         /* Cache things in the ixa without any refhold */
868         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
869         ixa->ixa_cred = cr;
870         ixa->ixa_cpid = pid;

872         if (is_system_labeled()) {
873                 ip_xmit_attr_restore_tsl(ixa, ixa->ixa_cred);
874         }

876         if (connp->conn_family == AF_INET6) {
877                 if (!IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
878                         error = dccp_connect_ipv6(dccp, &sin6->sin6_addr,
879                             sin6->sin6_port, sin6->sin6_flowinfo,
880                             sin6->__sin6_src_id, sin6->sin6_scope_id);
881                 } else {
882                         /*
883                          * Destination adress is mapped IPv6 address.
884                          * Source bound address should be unspecified or
885                          * IPv6 mapped address as well.
886                          */
887                         if (!IN6_IS_ADDR_UNSPECIFIED(
888                             &connp->conn_bound_addr_v6) &&
889                             !IN6_IS_ADDR_V4MAPPED(&connp->conn_bound_addr_v6)) {
890                                 return (EADDRNOTAVAIL);
891                         }

893                         dstaddrp = &V4_PART_OF_V6((sin6->sin6_addr));
894                         dstport = sin6->sin6_port;
895                         srcid = sin6->__sin6_src_id;
896                         error = dccp_connect_ipv4(dccp, dstaddrp, dstport,
897                             srcid);
898                 }
899         } else {
900                 dstaddrp = &sin->sin_addr.s_addr;
901                 dstport = sin->sin_port;
902                 srcid = 0;
903                 error = dccp_connect_ipv4(dccp, dstaddrp, dstport, srcid);
904         }

906         if (error != 0) {
907                 goto connect_failed;
908         }

910         /* XXX cluster */

912         DCCPS_BUMP_MIB(dccps, dccpActiveOpens);
913         dccp->dccp_active_open = 1;

915         DTRACE_DCCP6(state__change, void, NULL, ip_xmit_attr_t *,
916             connp->conn_ixa, void, NULL, dccp_t *, dccp, void, NULL,
917             int32_t, DCCPS_BOUND);

919         req_mp = dccp_generate_request(connp);
```

```
920         if (req_mp != NULL) {
921                 /*
922                  * We must bump the generation before sending the request
923                  * to ensure that we use the right generation in case
924                  * this thread issues a "connected" up call.
925                  */
926                 SOCK_CONNID_BUMP(dccp->dccp_connid);
927
928                 DTRACE_DCCP5(connect__request, mblk_t *, NULL,
929                     ip_xmit_attr_t *, connp->conn_ixa,
930                     void_ip_t *, req_mp->b_rptr, dccp_t *, dccp,
931                     dccpha_t *,
932                     &req_mp->b_rptr[connp->conn_ixa->ixa_ip_hdr_length]);
933
934                 dccp_send_data(dccp, req_mp);
935         }
936
937         return (0);
938
939 connect_failed:
940         cmn_err(CE_NOTE, "dccp_do_connect failed");
941
942         connp->conn_faddr_v6 = ipv6_all_zeros;
943         connp->conn_fport = 0;
944         dccp->dccp_state = oldstate;
945
946         return (error);
947 }
948
949 /*
950  * Init values of a connection.
951  */
952 void
953 dccp_init_values(dccp_t *dccp, dccp_t *parent)
954 {
955         conn_t          *connp = dccp->dccp_connp;
956         dccp_stack_t    *dccps = dccp->dccp_dccps;
957
958         connp->conn_mlp_type = mlptSingle;
959 }
960
961 void *
962 dccp_get_conn(void *arg, dccp_stack_t *dccps)
963 {
964         dccp_t          *dccp = NULL;
965         conn_t          *connp;
966         squeue_t        *sqp = (squeue_t *)arg;
967         netstack_t      *ns;
968
969         /* XXX timewait */
970
971         connp = ipcl_conn_create(IPCL_DCCPCONN, KM_NOSLEEP,
972             dccps->dccps_netstack);
973         if (connp == NULL) {
974                 return (NULL);
975         }
976
977         dccp = connp->conn_dccp;
978         dccp->dccp_dccps = dccps;
979
980         /* List of features being negotated */
981         list_create(&dccp->dccp_features, sizeof (dccp_feature_t),
982             offsetof(dccp_feature_t, df_next));
983
984         connp->conn_recv = dccp_input_data;
985         connp->conn_recvicmp = dccp_icmp_input;
```



```
986         connp->conn_verifyicmp = dccp_verifyicmp;
987
988         connp->conn_ixa->ixa_notify = dccp_notify;
989         connp->conn_ixa->ixa_notify_cookie = dccp;
990
991         return ((void *)connp);
992 }
993
994 /*
995  * IPv4 connect.
996  */
997 static int
998 dccp_connect_ipv4(dccp_t *dccp, ipaddr_t *dstaddrp, in_port_t dstport,
999     uint_t srcid)
1000 {
1001         conn_t          *connp = dccp->dccp_connp;
1002         dccp_stack_t    *dccps = dccp->dccp_dccps;
1003         ipaddr_t        dstaddr = *dstaddrp;
1004         uint16_t        lport;
1005         int             error;
1006
1007         cmn_err(CE_NOTE, "dccp.c: dccp_connect_ipv4");
1008
1009         ASSERT(connp->conn_ipversion == IPV4_VERSION);
1010
1011         if (dstaddr == INADDR_ANY) {
1012                 dstaddr = htonl(INADDR_LOOPBACK);
1013                 *dstaddrp = dstaddr;
1014         }
1015
1016         if (srcid != 0 && connp->conn_laddr_v4 == INADDR_ANY) {
1017                 ip_srcid_find_id(srcid, &connp->conn_laddr_v6,
1018                     IPCL_ZONEID(connp), dccps->dccps_netstack);
1019                 connp->conn_saddr_v6 = connp->conn_laddr_v6;
1020         }
1021
1022         IN6_IPADDR_TO_V4MAPPED(dstaddr, &connp->conn_faddr_v6);
1023         connp->conn_fport = dstport;
1024
1025         if (dccp->dccp_state == DCCPS_CLOSED) {
1026                 lport = dccp_update_next_port(dccps->dccps_next_port_to_try,
1027                     dccp, B_TRUE);
1028                 lport = dccp_bindi(dccp, lport, &connp->conn_laddr_v6, 0,
1029                     B_TRUE, B_FALSE, B_FALSE);
1030
1031                 if (lport == 0) {
1032                         return (-TNOADDR);
1033                 }
1034         }
1035
1036         error = dccp_set_destination(dccp);
1037         if (error != 0) {
1038                 return (error);
1039         }
1040
1041         /*
1042          * Don't connect to oneself.
1043          */
1044         if (connp->conn_faddr_v4 == connp->conn_laddr_v4 &&
1045             connp->conn_fport == connp->conn_lport) {
1046                 return (-TBADADDR);
1047         }
1048
1049         /* XXX state */
1050
1051         return (ipcl_conn_insert_v4(connp));
```

```
1052 }

1054 /*
1055  * IPv6 connect.
1056  */
1057 static int
1058 dccp_connect_ipv6(dccp_t *dccp, in6_addr_t *dstaddrp, in_port_t dstport,
1059     uint32_t flowinfo, uint_t srcid, uint32_t scope_id)
1060 {
1061         cmn_err(CE_NOTE, "dccp.c: dccp_connect_ipv6");

1063         return (0);
1064 }

1066 /*
1067  * Set the ports via conn_connect and build the template
1068  * header.
1069  */
1070 int
1071 dccp_set_destination(dccp_t *dccp)
1072 {
1073         conn_t          *connp = dccp->dccp_connp;
1074         dccp_stack_t    *dccps = dccp->dccp_dccps;
1075         iulp_t          uinfo;
1076         uint32_t        flags;
1077         int             error;

1079         flags = IPDF_LSO | IPDF_ZCOPY;
1080         flags |= IPDF_UNIQUE_DCE;

1082         mutex_enter(&connp->conn_lock);
1083         error = conn_connect(connp, &uinfo, flags);
1084         mutex_exit(&connp->conn_lock);
1085         if (error != 0) {
1086                 cmn_err(CE_NOTE, "conn_connect failed");
1087                 return (error);
1088         }

1090         error = dccp_build_hdrs(dccp);
1091         if (error != 0) {
1092                 cmn_err(CE_NOTE, "dccp_build_hdrs failed");
1093                 return (error);
1094         }

1096         /* XXX */

1098         /* Initialise the ISS */
1099         dccp_iss_init(dccp);

1101         mutex_enter(&connp->conn_lock);
1102         connp->conn_state_flags &= ~CONN_INCIPIENT;
1103         mutex_exit(&connp->conn_lock);

1105         return (0);
1106 }

1108 /*
1109  * Init the ISS.
1110  */
1111 static void
1112 dccp_iss_init(dccp_t *dccp)
1113 {
1114         cmn_err(CE_NOTE, "dccp.c: dccp_iss_init");

1116         dccp->dccp_iss += gethrtime();
1117 }
```



```
1118 #endif /* ! codereview */
```

```
**********************************************************
     913 Mon Jul  9 14:38:12 2012
new/usr/src/uts/common/inet/dccp/dccp.conf
dccp: starting module template
**********************************************************
```

```
   1 #
   2 # CDDL HEADER START
   3 #
   4 # The contents of this file are subject to the terms of the
   5 # Common Development and Distribution License, Version 1.0 only
   6 # (the "License").  You may not use this file except in compliance
   7 # with the License.
   8 #
   9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10 # or http://www.opensolaris.org/os/licensing.
  11 # See the License for the specific language governing permissions
  12 # and limitations under the License.
  13 #
  14 # When distributing Covered Code, include this CDDL HEADER in each
  15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16 # If applicable, add the following below this CDDL HEADER, with the
  17 # fields enclosed by brackets "[]" replaced with your own identifying
  18 # information: Portions Copyright [yyyy] [name of copyright owner]
  19 #
  20 # CDDL HEADER END
  21 #
  22 #
  23 # Copyright (c) 1992, by Sun Microsystems, Inc.
  24 #

  26 name="dccp" parent="pseudo" instance=0;
  27 #endif /* ! codereview */
```

```
**********************************************************
     914 Mon Jul  9 14:38:12 2012
new/usr/src/uts/common/inet/dccp/dccp6.conf
dccp: clean up
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License, Version 1.0 only
# (the "License").  You may not use this file except in compliance
# with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright (c) 1992, by Sun Microsystems, Inc.
#

name="dccp6" parent="pseudo" instance=0;
#endif /* ! codereview */
```

```
**********************************************************
    1578 Mon Jul  9 14:38:12 2012
new/usr/src/uts/common/inet/dccp/dccp6ddi.c
dccp: clean up
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/modctl.h>
#include <inet/common.h>
#include <inet/ip.h>

#define INET_NAME       "dccp6"
#define INET_DEVSTRTAB  dccpinfov6
#define INET_DEVDESC    "DCCP6 STREAMS driver"
#define INET_DEVMINOR   0
#define INET_DEVMTFLAGS (D_MP|_D_DIRECT)

#include "../inetddi.c"

int
_init(void)
{
        /*
         * device initialization happens when the actual code containing
         * module (/kernel/drv/ip) is loaded, and driven from ip_ddi_init()
         */
        return (mod_install(&modlinkage));
}

int
_fini(void)
{
        return (mod_remove(&modlinkage));
}

int
_info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}
#endif /* ! codereview */
```

```
**********************************************************
   10340 Mon Jul  9 14:38:12 2012
new/usr/src/uts/common/inet/dccp/dccp_bind.c
dccp: bind function
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * This file contains function related to binding.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/proto_set.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"

/* Setable in /etc/system */
static uint32_t dccp_random_anon_port = 1;

static int dccp_bind_select_lport(dccp_t *, in_port_t *, boolean_t,
    cred_t *);
```



```
void
dccp_bind_hash_insert(dccp_df_t *tbf, dccp_t *dccp, int caller_holds_lock)
{
        conn_t  *connp = dccp->dccp_connp;
        conn_t  *connext;
        dccp_t  **dccpp;
        dccp_t  *dccpnext;
        dccp_t  *dccphash;

        cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_hash_insert");

        /* XXX:DCCP */

        dccpp = &tbf->df_dccp;
        if (!caller_holds_lock) {
                mutex_enter(&tbf->df_lock);
        } else {
                ASSERT(MUTEX_HELD(&tbf->df_lock));
        }
        dccphash = dccpp[0];
        dccpnext = NULL;

        if (dccphash != NULL) {
                /* XXX:DCCP */
        }

insert:
        dccp->dccp_bind_hash_port = dccpnext;
        dccp->dccp_bind_hash = dccphash;
        dccp->dccp_ptpbhn = dccpp;
        dccpp[0] = dccp;

        if (!caller_holds_lock) {
                mutex_exit(&tbf->df_lock);
        }
}

void
dccp_bind_hash_remove(dccp_t *dccp)
{
}

/*
 * Check for a valid address and get a local port.
 */
int
dccp_bind_check(conn_t *connp, struct sockaddr *sa, socklen_t len, cred_t *cr,
    boolean_t bind_to_req_port_only)
{
        dccp_t          *dccp = connp->conn_dccp;
        ip_stack_t      *ips = connp->conn_netstack->netstack_ip;
        ip_xmit_attr_t  *ixa = connp->conn_ixa;
        sin_t           *sin;
        sin6_t          *sin6;
        ipaddr_t        v4addr;
        in6_addr_t      v6addr;
        ip_laddr_t      laddr_type = IPVL_UNICAST_UP;
        zoneid_t        zoneid = IPCL_ZONEID(connp);
        in_port_t       requested_port;
        uint_t          scopeid = 0;
        int             error;

        cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_check");

        ASSERT((uintptr_t)len <= (uintptr_t)INT_MAX);
```

```
129         /*
130          * We should be in a pre-listen state.
131          */
132         if (dccp->dccp_state == DCCPS_LISTEN) {
133                 return (0);
134         } else if (dccp->dccp_state > DCCPS_LISTEN) {
135                 if (connp->conn_debug) {
136                         (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
137                             "dccp_bind: bad state, %d", dccp->dccp_state);
138                 }
139                 return (-TOUTSTATE);
140         }
141
142         /*
143          * Check for a valid address parameter. Then validate the
144          * addresses and copy them and the required port in.
145          */
146         ASSERT(sa != NULL && len != 0);
147         if (!OK_32PTR((char *)sa)) {
148                 if (connp->conn_debug) {
149                         (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
150                             "dccp_bind: bad address parameter, "
151                             "address %p, len %d", (void *)sa, len);
152                 }
153                 return (-TPROTO);
154         }
155
156         error = proto_verify_ip_addr(connp->conn_family, sa, len);
157         if (error != 0) {
158                 return (error);
159         }
160
161         switch (len) {
162         case sizeof (sin_t):
163                 sin = (sin_t *)sa;
164                 v4addr = sin->sin_addr.s_addr;
165                 requested_port = ntohs(sin->sin_port);
166                 IN6_IPADDR_TO_V4MAPPED(v4addr, &v6addr);
167                 if (v4addr != INADDR_ANY) {
168                         laddr_type = ip_laddr_verify_v4(v4addr, zoneid, ips,
169                             B_FALSE);
170                 }
171                 break;
172
173         case sizeof (sin6_t):
174                 sin6 = (sin6_t *)sa;
175                 v6addr = sin6->sin6_addr;
176                 requested_port = ntohs(sin6->sin6_port);
177                 if (IN6_IS_ADDR_V4MAPPED(&v6addr)) {
178                         if (connp->conn_ipv6_v6only) {
179                                 return (EADDRNOTAVAIL);
180                         }
181
182                         IN6_V4MAPPED_TO_IPADDR(&v6addr, v4addr);
183                         if (v4addr != INADDR_ANY) {
184                                 laddr_type = ip_laddr_verify_v4(v4addr, zoneid,
185                                     ips, B_FALSE);
186                         }
187                 } else {
188                         if (!IN6_IS_ADDR_UNSPECIFIED(&v6addr)) {
189                                 if (IN6_IS_ADDR_LINKSCOPE(&v6addr)) {
190                                         scopeid = sin6->sin6_scope_id;
191                                         laddr_type = ip_laddr_verify_v6(&v6addr,
192                                             zoneid, ips, B_FALSE, scopeid);
193                                 }
```



```
194                         }
195                 }
196                 break;
197
198         default:
199                 if (connp->conn_debug) {
200                         (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
201                             "dccp_bind: bad address length, %d", len);
202                 }
203                 return (EAFNOSUPPORT);
204         }
205
206         if (laddr_type == IPVL_BAD) {
207                 return (EADDRNOTAVAIL);
208         }
209
210         connp->conn_bound_addr_v6 = v6addr;
211         if (scopeid != 0) {
212                 ixa->ixa_flags |= IXAF_SCOPEID_SET;
213                 ixa->ixa_scopeid = scopeid;
214                 connp->conn_incoming_ifindex = scopeid;
215         } else {
216                 ixa->ixa_flags &= ~IXAF_SCOPEID_SET;
217                 connp->conn_incoming_ifindex = connp->conn_bound_if;
218         }
219
220         connp->conn_laddr_v6 = v6addr;
221         connp->conn_saddr_v6 = v6addr;
222
223         bind_to_req_port_only = requested_port != 0 && bind_to_req_port_only;
224
225         error = dccp_bind_select_lport(dccp, &requested_port,
226             bind_to_req_port_only, cr);
227         if (error != 0) {
228                 connp->conn_laddr_v6 = ipv6_all_zeros;
229                 connp->conn_saddr_v6 = ipv6_all_zeros;
230                 connp->conn_bound_addr_v6 = ipv6_all_zeros;
231         }
232
233         return (error);
234 }
235
236 /*
237  * Bind to a local port.
238  */
239 static int
240 dccp_bind_select_lport(dccp_t *dccp, in_port_t *requested_port_ptr,
241     boolean_t bind_to_req_port_only, cred_t *cr)
242 {
243         dccp_stack_t    *dccps = dccp->dccp_dccps;
244         conn_t          *connp = dccp->dccp_connp;
245         zone_t          *zone;
246         in_port_t       allocated_port;
247         in_port_t       requested_port = *requested_port_ptr;
248         in6_addr_t      v6addr = connp->conn_laddr_v6;
249         boolean_t       user_specified;
250
251         cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_select_lport");
252
253         ASSERT(cr != NULL);
254
255         if (requested_port == 0) {
256                 requested_port =
257                     dccp_update_next_port(dccps->dccps_next_port_to_try,
258                     dccp, B_TRUE);
259                 if (requested_port == 0) {
```

```
260                         return (-TNOADDR);
261                 }
262                 user_specified = B_FALSE;

264         }  else {
265                 int             i;
266                 boolean_t       priv = B_FALSE;

268                 if (requested_port < dccps->dccps_smallest_nonpriv_port) {
269                         priv = B_TRUE;
270                 } else {
271                         for (i = 0; i < dccps->dccps_num_epriv_ports; i++) {
272                                 if (requested_port ==
273                                     dccps->dccps_epriv_ports[i]) {
274                                         priv = B_TRUE;
275                                         break;
276                                 }
277                         }
278                 }

280                 if (priv) {
281                         if (secpolicy_net_privaddr(cr, requested_port,
282                             IPPROTO_DCCP) != 0) {
283                                 if (connp->conn_debug) {
284                                         (void) strlog(DCCP_MOD_ID, 0, 1,
285                                             SL_ERROR|SL_TRACE,
286                                             "tcp_bind: no priv for port %d",
287                                             requested_port);
288                                 }
289                                 return (-TACCES);
290                         }
291                 }

293                 user_specified = B_TRUE;
294                 //connp = dccp->dccp_connp;

296                 /* XXX */
297         }

299         allocated_port = dccp_bindi(dccp, requested_port, &v6addr,
300             connp->conn_reuseaddr, B_FALSE, bind_to_req_port_only,
301             user_specified);

303         if (allocated_port == 0) {
304                 /* XXX */
305                 if (bind_to_req_port_only) {
306                         if (connp->conn_debug) {
307                                 (void) strlog(DCCP_MOD_ID, 0, 1,
308                                     SL_ERROR|SL_TRACE,
309                                     "dccp_bind: requested addr busy");
310                         }
311                         return (-TADDRBUSY);
312                 } else {
313                         if (connp->conn_debug) {
314                                 (void) strlog(DCCP_MOD_ID, 0, 1,
315                                     SL_ERROR|SL_TRACE,
316                                     "dccp_bind: out of ports?");
317                         }
318                         return (-TNOADDR);
319                 }
320         }

322         *requested_port_ptr = allocated_port;
323         return (0);
324 }
```



```
326 in_port_t
327 dccp_bindi(dccp_t *dccp, in_port_t port, const in6_addr_t *laddr,
328     int reuseaddr, boolean_t quick_connect, boolean_t bind_to_req_port_only,
329     boolean_t user_specified)
330 {
331         dccp_stack_t    *dccps = dccp->dccp_dccps;
332         conn_t          *connp = dccp->dccp_connp;
333         int             count = 0;
334         int             loopmax;

336         cmn_err(CE_NOTE, "dccp_bind.c: dccp_bindi");

338         if (bind_to_req_port_only) {
339                 loopmax = 1;
340         } else {
341                 if (connp->conn_anon_priv_bind) {
342                         loopmax = IPPORT_RESERVED -
343                             dccps->dccps_min_anonpriv_port;
344                 } else {
345                         loopmax = (dccps->dccps_largest_anon_port -
346                             dccps->dccps_smallest_anon_port + 1);
347                 }
348         }

350         do {
351                 conn_t          *lconnp;
352                 dccp_t          *ldccp;
353                 dccp_df_t       *ldf;
354                 uint16_t        lport;

356                 lport = htons(port);

358                 dccp_bind_hash_remove(dccp);
359                 ldf = &dccps->dccps_bind_fanout[DCCP_BIND_HASH(lport,
360                     dccps->dccps_bind_fanout_size)];
361                 mutex_enter(&ldf->df_lock);
362                 for (ldccp = ldf->df_dccp; ldccp != NULL;
363                     ldccp = ldccp->dccp_bind_hash) {
364                         if (lport == ldccp->dccp_connp->conn_lport) {
365                                 break;
366                         }
367                 }

369                 if (ldccp != NULL) {
370                         mutex_exit(&ldf->df_lock);
371                 } else {
372                         dccp->dccp_state = DCCPS_BOUND;

374                         connp->conn_lport = htons(port);

376                         ASSERT(&dccps->dccps_bind_fanout[DCCP_BIND_HASH(
377                              connp->conn_lport,
378                              dccps->dccps_bind_fanout_size)] == ldf);
379                         dccp_bind_hash_insert(ldf, dccp, 1);

381                         mutex_exit(&ldf->df_lock);

383                         if (user_specified) {
384                                 return (port);
385                         }

387                         if (!connp->conn_anon_priv_bind) {
388                                 dccps->dccps_next_port_to_try = port + 1;
389                         }

391                         return (port);
```

```
392                 }
394                 if (port == 0) {
395                         break;
396                 }
398         } while (++count < loopmax);
400         cmn_err(CE_NOTE, "dccp_bind.c: dccp_bindi exit");
402         return (0);
403 }
405 in_port_t
406 dccp_update_next_port(in_port_t port, const dccp_t *dccp, boolean_t random)
407 {
408         dccp_stack_t    *dccps = dccp->dccp_dccps;
409         boolean_t       restart = B_FALSE;
410         int             i;
412         cmn_err(CE_NOTE, "dccp_bind.c: dccp_update_next_port");
414         if (random && dccp_random_anon_port != 0) {
415                 (void) random_get_pseudo_bytes((uint8_t *)&port,
416                     sizeof (in_port_t));
418                 if (port < dccps->dccps_smallest_anon_port) {
419                         port = dccps->dccps_smallest_anon_port +
420                             port % (dccps->dccps_largest_anon_port -
421                             dccps->dccps_smallest_anon_port);
422                 }
423         }
425 retry:
426         if (port < dccps->dccps_smallest_anon_port) {
427                 port = (in_port_t)dccps->dccps_smallest_anon_port;
428         }
430         if (port > dccps->dccps_largest_anon_port) {
431                 if (restart) {
432                         return (0);
433                 }
434                 restart = B_TRUE;
435                 port = (in_port_t)dccps->dccps_smallest_anon_port;
436         }
438         if (port < dccps->dccps_smallest_nonpriv_port) {
439                 port = (in_port_t)dccps->dccps_smallest_nonpriv_port;
440         }
442         for (i = 0; i < dccps->dccps_num_epriv_ports; i++) {
443                 if (port == dccps->dccps_epriv_ports[i]) {
444                         port++;
445                         goto retry;
446                 }
447         }
449         return (port);
450 }
451 #endif /* ! codereview */
```

```
**********************************************************
    6296 Mon Jul  9 14:38:12 2012
new/usr/src/uts/common/inet/dccp/dccp_features.c
dccp: options and features
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner. All rights reserved.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/debug.h>
#include <sys/cmn_err.h>

#include "dccp_impl.h"
#include "dccp_stack.h"

/*
 * This file contains functions to parse and process DCCP options
 * and features.
 */

static void     dccp_parse_feature(dccp_t *, uint8_t, uint8_t, uchar_t *,
                    boolean_t);

/*
 * Feature handling.
 */
static void
dccp_parse_feature(dccp_t *dccp, uint8_t option, uint8_t length, uchar_t *up,
    boolean_t mandatory)
{
        dccp_feature_t  *feature;
        uint8_t         feature_type;
        uint8_t         feature_length = length - 1;

        cmn_err(CE_NOTE, "dccp_features.c: dccp_parse_feature");

        feature_type = *up;

        switch (feature_type) {
        case DCCP_FEATURE_CCID:
                cmn_err(CE_NOTE, "DCCP_FEATURE_CCID");
                break;
        case DCCP_FEATURE_ALLOW_SHORT_SEQNOS:
                cmn_err(CE_NOTE, "DCCP_FEATURE_ALLOW_SHORT_SEQNOS");
                break;
        case DCCP_FEATURE_SEQUENCE_WINDOW:
                cmn_err(CE_NOTE, "DCCP_FEATURE_SEQUENCE_WINDOW");
                break;
        case DCCP_FEATURE_ECN_INCAPABLE:
                cmn_err(CE_NOTE, "DCCP_FEATURE_ECN_INCAPABLE");
                break;
        case DCCP_FEATURE_ACK_RATIO:
                cmn_err(CE_NOTE, "DCCP_FEATURE_ACK_RATIO");
```



```
                break;
        case DCCP_FEATURE_SEND_ACK_VECTOR:
                cmn_err(CE_NOTE, "DCCP_FEATURE_SEND_ACK_VECTOR");
                break;
        case DCCP_FEATURE_SEND_NDP_COUNT:
                cmn_err(CE_NOTE, "DCCP_FEATURE_SEND_NDP_COUNT");
                break;
        case DCCP_FEATURE_MIN_CHECKSUM_COVERAGE:
                cmn_err(CE_NOTE, "DCCP_FEATURE_MIN_CHECKSUM_COVERAGE");
                break;
        case DCCP_FEATURE_CHECK_DATA_CHECKSUM:
                cmn_err(CE_NOTE, "DCCP_FEATURE_CHECK_DATA_CHECKSUM");
                break;

        default:
                cmn_err(CE_NOTE, "ERROR DEFAULT");
                break;
        }

        feature = (dccp_feature_t *)kmem_alloc(sizeof (dccp_feature_t),
            KM_SLEEP);
        if (feature == NULL) {
                return;
        }

        feature->df_option = option;
        feature->df_type = feature_type;
        feature->df_mandatory = mandatory;

        list_insert_tail(&dccp->dccp_features, feature);
}

/*
 * Parse the options in a DCCP header.
 */
void
dccp_parse_options(dccp_t *dccp, dccpha_t *dccpha)
{
        uchar_t         *end;
        uchar_t         *up;
        uint8_t         dccp_type;
        uint8_t         option_type;
        uint8_t         option_length;
        int             len;
        int             i;
        uchar_t         *value;
        boolean_t       mandatory = B_FALSE;

        cmn_err(CE_NOTE, "dccp_features.c: dccp_parse_options");

        dccp_type = dccpha->dha_type;

        up = (uchar_t *)dccpha;
        end = up + DCCP_HDR_LENGTH(dccpha);
        up += 20;

        if (dccp_type != DCCP_PKT_REQUEST) {
                cmn_err(CE_NOTE, "not request pkt");
                return;
        }

        while (up != end) {
                option_length = 0;
                option_type = *up++;

                if (option_type > 31) {
```

```
128                         option_length = *up++;
129                         option_length -= 2;
130                         value = up;

132                         up += option_length;
133                 }

135                 switch (option_type) {
136                 case DCCP_OPTION_PADDING:
137                         cmn_err(CE_NOTE, "PADDING");
138                         break;
139                 case DCCP_OPTION_MANDATORY:
140                         cmn_err(CE_NOTE, "MANDATORY");
141                         mandatory = B_TRUE;
142                         break;
143                 case DCCP_OPTION_SLOW_RECEIVER:
144                         cmn_err(CE_NOTE, "SLOW RECEIVER");
145                         break;
146                 case DCCP_OPTION_CHANGE_L:
147                 case DCCP_OPTION_CONFIRM_L:
148                 case DCCP_OPTION_CHANGE_R:
149                 case DCCP_OPTION_CONFIRM_R:
150                         dccp_parse_feature(dccp, option_type, option_length,
151                             value, mandatory);
152                         break;
153                 case DCCP_OPTION_INIT_COOKIE:
154                         cmn_err(CE_NOTE, "INIT COOKIE");
155                         break;
156                 case DCCP_OPTION_NDP_COUNT:
157                         cmn_err(CE_NOTE, "NDP COUNT");
158                         break;
159                 case DCCP_OPTION_ACK_VECTOR_1:
160                         cmn_err(CE_NOTE, "ACK VECTOR 1");
161                         break;
162                 case DCCP_OPTION_ACK_VECTOR_2:
163                         cmn_err(CE_NOTE, "ACK VECTOR 2");
164                         break;
165                 case DCCP_OPTION_DATA_DROPPED:
166                         cmn_err(CE_NOTE, "DATA DROPPED");
167                         break;
168                 case DCCP_OPTION_TIMESTAMP:
169                         cmn_err(CE_NOTE, "TIMESTAMP");
170                         if (option_length == 4) {
171                                 cmn_err(CE_NOTE, "timestamp length == 4");
172                         } else {
173                                 cmn_err(CE_NOTE, "timestamp length != 4");
174                         }
175                         break;
176                 case DCCP_OPTION_TIMESTAMP_ECHO:
177                         cmn_err(CE_NOTE, "TIMESTAMP ECHO");
178                         break;
179                 case DCCP_OPTION_ELAPSED_TIME:
180                         cmn_err(CE_NOTE, "ELAPSES TIME");
181                         break;
182                 case DCCP_OPTION_DATA_CHECKSUM:
183                         cmn_err(CE_NOTE, "DATA CHECKSUM");
184                         break;

186                 default:
187                         cmn_err(CE_NOTE, "DEFAULT");
188                         break;
189                 }

191                 if (option_type != DCCP_OPTION_MANDATORY) {
192                         mandatory = B_FALSE;
193                 }
```



```
194         }
195 }

197 void
198 dccp_process_options(dccp_t *dccp, dccpha_t *dccpha)
199 {
200         cmn_err(CE_NOTE, "dccp_features.c: dccp_process_features");

202         dccp_parse_options(dccp, dccpha);
203 }

205 int
206 dccp_generate_options(dccp_t *dccp, void **opt, size_t *opt_len)
207 {
208         dccp_feature_t  *feature = NULL;
209         uint8_t         buf[1024];      /* XXX */
210         uint8_t         option_type;
211         uint_t          len = 0;
212         uint_t          total_len;
213         void            *options;
214         int             rest;

216         cmn_err(CE_NOTE, "dccp_features.c: dccp_generate_options");

218         for (feature = list_head(&dccp->dccp_features); feature;
219             feature = list_next(&dccp->dccp_features, feature)) {
220                 if (feature->df_option == DCCP_OPTION_CHANGE_L) {
221                         option_type = DCCP_OPTION_CONFIRM_R;
222                 } else {
223                         option_type = DCCP_OPTION_CONFIRM_L;
224                 }
225 /*
226                 if (feature->df_mandatory == B_TRUE) {
227                         buf[len] = DCCP_OPTION_MANDATORY;
228                         len++;
229                 }
230 */
231                 if (feature->df_type == DCCP_FEATURE_CCID) {
232                         cmn_err(CE_NOTE, "FOUND DCCP_FEATURE_CCID");

234                         buf[len] = option_type;
235                         len++;
236                         buf[len] = 4;
237                         len++;
238                         buf[len] = DCCP_FEATURE_CCID;
239                         len++;
240                         buf[len] = 2;
241                         len++;
242                 }

244                 if (feature->df_type == DCCP_FEATURE_ALLOW_SHORT_SEQNOS) {
245                         buf[len] = option_type;
246                         len++;
247                         buf[len] = 4;
248                         len++;
249                         buf[len] = feature->df_type;
250                         len++;
251                         buf[len] = 0;
252                         len++;
253                 }

255                 if (feature->df_type == DCCP_FEATURE_ECN_INCAPABLE) {
256                         buf[len] = option_type;
257                         len++;
258                         buf[len] = 4;
259                         len++;
```

```
260                         buf[len] = feature->df_type;
261                         len++;
262                         buf[len] = 1;
263                         len++;
264                 }
265         }

267         total_len = ((len + (4 - 1)) / 4) * 4;
268         options = kmem_zalloc(total_len, KM_SLEEP);
269         if (options == NULL) {
270                 cmn_err(CE_NOTE, "kmem_zalloc failed");
271                 return (ENOMEM);
272         }
273         memcpy(options, buf, len);

275         *opt = options;
276         *opt_len = len;

278         return (0);
279 }

281 void
282 dccp_features_init(void)
283 {
284 }

286 void
287 dccp_features_destroy(void)
288 {
289 }
290 #endif /* ! codereview */
```

```
**********************************************************
   10871 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_impl.h
dccp: starting module template
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner. All rights reserved.
 */

#ifndef _INET_DCCP_IMPL_H
#define _INET_DCCP_IMPL_H

#include <sys/int_types.h>
#include <sys/netstack.h>
#include <sys/socket.h>
#include <sys/socket_proto.h>

#include <netinet/in.h>
#include <netinet/ip6.h>
#include <netinet/dccp.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/optcom.h>
#include <inet/tunables.h>

#include "dccp_stack.h"

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef  _KERNEL

#define DCCP_MOD_ID             5999 /* XXX */

extern struct qinit     dccp_sock_winit;
extern struct qinit     dccp_winit;

extern sock_downcalls_t sock_dccp_downcalls;

#define DCCP_XMIT_LOWATER       (4 * 1024)
#define DCCP_XMIT_HIWATER       49152
#define DCCP_RECV_LOWATER       (2 * 1024)
#define DCCP_RECV_HIWATER       128000

#define TIDUSZ  4096    /* transport interface data unit size */

/*
 * Bind hash array size and hash function.
 */
#define DCCP_BIND_FANOUT_SIZE           128
#define DCCP_BIND_HASH(lport, size)     ((ntohs((uint16_t)lport)) & (size - 1))
```



```
/*
 * Was this tcp created via socket() interface?
 */
#define DCCP_IS_SOCKET(dccp)    ((dccp)->dccp_issocket)

#define DCCP_HDR_LENGTH(dccph)          (((dccph_t *)dccph)->dh_offset * 4) /* X
#define DCCP_MAX_HDR_LENGTH             1020
#define DCCP_MIN_HEADER_LENGTH          12

/* Packet types (RFC 4340, Section 5.1.) */
#define DCCP_PKT_REQUEST        0
#define DCCP_PKT_RESPONSE       1
#define DCCP_PKT_DATA           2
#define DCCP_PKT_ACK            3
#define DCCP_PKT_DATAACK        4
#define DCCP_PKT_CLOSEREQ       5
#define DCCP_PKT_CLOSE          6
#define DCCP_PKT_RESET          7
#define DCCP_PKT_SYNC           8
#define DCCP_PKT_SYNCACK        9

/*
 * DCCP options and features.
 */

/* Options types (RFC 4340, Section 5.8.) */
#define DCCP_OPTION_PADDING             0
#define DCCP_OPTION_MANDATORY           1
#define DCCP_OPTION_SLOW_RECEIVER       2
#define DCCP_OPTION_CHANGE_L            32
#define DCCP_OPTION_CONFIRM_L           33
#define DCCP_OPTION_CHANGE_R            34
#define DCCP_OPTION_CONFIRM_R           35
#define DCCP_OPTION_INIT_COOKIE         36
#define DCCP_OPTION_NDP_COUNT           37
#define DCCP_OPTION_ACK_VECTOR_1        38
#define DCCP_OPTION_ACK_VECTOR_2        39
#define DCCP_OPTION_DATA_DROPPED        40
#define DCCP_OPTION_TIMESTAMP           41
#define DCCP_OPTION_TIMESTAMP_ECHO      42
#define DCCP_OPTION_ELAPSED_TIME        43
#define DCCP_OPTION_DATA_CHECKSUM       44

/* Feature types (RFC 4340, Section 6.4.) */
#define DCCP_FEATURE_CCID                       1
#define DCCP_FEATURE_ALLOW_SHORT_SEQNOS         2
#define DCCP_FEATURE_SEQUENCE_WINDOW            3
#define DCCP_FEATURE_ECN_INCAPABLE              4
#define DCCP_FEATURE_ACK_RATIO                  5
#define DCCP_FEATURE_SEND_ACK_VECTOR            6
#define DCCP_FEATURE_SEND_NDP_COUNT             7
#define DCCP_FEATURE_MIN_CHECKSUM_COVERAGE      8
#define DCCP_FEATURE_CHECK_DATA_CHECKSUM        9

/* Feature negotation states (RFC 4340, Section 6.6.2.) */
#define DCCP_FEATURE_STATE_CHANGING             0
#define DCCP_FEATURE_STATE_UNSTABLE             1
#define DCCP_FEATURE_STATE_STABLE               2

typedef struct dccp_feature_s {
        list_node_t     df_next;
        uint8_t         df_option;
        uint8_t         df_type;
        uint8_t         df_state;
        uint64_t        df_value;
        boolean_t       df_mandatory;
```

```
} dccp_feature_t;

/* Options in DCCP header */
typedef struct dccp_opt_s {
        int             type;
        boolean_t       mandatory;
} dccp_opt_t;

/*
 * DCCP header structures.
 */

/* Generic protocol header (RFC 4340, Section 5.1.) */
typedef struct dccphdr_s {
        uint8_t         dh_lport[2];
        uint8_t         dh_fport[2];
        uint8_t         dh_offset;
        uint8_t         dh_ccval:4,
                        dh_cscov:4;
        uint8_t         db_sum[2];
        uint8_t         dh_reserved:3,
                        dh_type:4,
                        dh_x:1;
        uint8_t         dh_res_seq;
        uint8_t         dh_seq[2];
} dccph_t;


/* Generic protocol header aligned (RFC 4340, Section 5.1.) */
typedef struct dccphdra_s {
        in_port_t       dha_lport;              /* Source port */
        in_port_t       dha_fport;              /* Destination port */
        uint8_t         dha_offset;             /* Data offset */
        uint8_t         dha_ccval:4,            /* */
                        dha_cscov:4;            /* */
        uint16_t        dha_sum;                /* Checksum */
        uint8_t         dha_x:1,                /* Reserved */
                        dha_type:4,             /* Packet type */
                        dha_reserved:3;         /* Header type */
        uint8_t         dha_res_seq;
        uint16_t        dha_seq;                /* Partial sequence number */
} dccpha_t;

typedef struct dccphdra_ext_s {
        uint32_t        dha_ext_seq;
} dccpha_ext_t;

/* Acknowledgement number */
typedef struct dccphdra_ack {
        uint16_t        dha_ack_reserved;
        uint16_t        dha_ack_high;
        uint32_t        dha_ack_low;
} dccpha_ack_t;

/* Service number */
typedef struct dccphdra_srv {
        uint32_t        dha_srv_code;
} dccpha_srv_t;

/* Reset data */
typedef struct dccphdra_reset {
        uint8_t         dha_reset_code;
        uint8_t         dha_reset_data[3];
} dccpha_reset_t;

/* Internal DCCP structure */
```



```
typedef struct dccp_s {

        conn_t          *dccp_connp;            /* Backpointer to conn_t */
        dccp_stack_t    *dccp_dccps;            /* Backpointer to dccp_stack_t *

        int32_t         dccp_state;

        uint32_t
                dccp_loopback: 1,       /* Src and dst are the same machine */
                dccp_localnet: 1,       /* Src and dst are on the same subnet */
                dccp_active_open: 1,    /* This is a active open */
                dccp_dummy: 1;

        /* Bind related */
        struct dccp_s   *dccp_bind_hash;        /* Bind hash chain */
        struct dccp_s   *dccp_bind_hash_port;   /* Bound to the same port */
        struct dccp_s   **dccp_ptpbhn;

        struct dccphdra_s *dccp_dccpha;         /* Template header */

        mblk_t          *dccp_xmit_head;

        /*
         * Pointers into the header template.
         */
        ipha_t          *dccp_ipha;
        ip6_t           *dccp_ip6h;

        t_uscalar_t     dccp_acceptor_id;       /* ACCEPTOR_id */

        sock_connid_t   dccp_connid;

        /* Incrementing pending conn req ID */
        t_scalar_t      dccp_conn_req_seqnum;

        boolean_t       dccp_issocket;          /* This is a socket dccp */

        /* List of features being negotiated */
        list_t          dccp_features;

        /*
         *Sequence numbers (Section 7.1.)
         */
        uint64_t        dccp_iss;       /* Initial sequence number sent */
        uint64_t        dccp_isr;       /* Initial sequence number received */
        uint64_t        dccp_gss;       /* Greatest sequence number sent */
        uint64_t        dccp_gsr;       /* Greatest sequence */
                                        /* number received */
        uint64_t        dccp_gar;       /* Greatest acknowledgement */
                                        /* number received */
} dccp_t;

#define dccps_smallest_nonpriv_port     dccps_propinfo_tbl[0].prop_cur_uval
#define dccps_smallest_anon_port        dccps_propinfo_tbl[1].prop_cur_uval
#define dccps_largest_anon_port         dccps_propinfo_tbl[2].prop_cur_uval

#define dccps_dbg                       dccps_propinfo_tbl[4].prop_cur_uval
#define dccps_rst_sent_rate_enabled     dccps_propinfo_tbl[5].prop_cur_uval
#define dccps_rst_sent_rate             dccps_propinfo_tbl[6].prop_cur_uval

typedef struct dccp_df_s {
        struct dccp_s   *df_dccp;
        kmutex_t        df_lock;
        uchar_t         df_pad[TF_CACHEL_PAD - (sizeof (dccp_t *) +
                            sizeof (kmutex_t))];
} dccp_df_t;
```

```
261 extern struct qinit dccp_rinitv4, dccp_rinitv6;

263 extern optdb_obj_t      dccp_opt_obj;
264 extern uint_t           dccp_max_optsize;

266 extern int              dccp_squeue_flag;

268 /*
269  * Functions in dccp.c
270  */
271 extern int      dccp_build_hdrs(dccp_t *);
272 extern conn_t   *dccp_create_common(cred_t *, boolean_t, boolean_t, int *);
273 extern void     dccp_close_common(conn_t *, int);
274 extern int      dccp_do_bind(conn_t *, struct sockaddr *, socklen_t, cred_t *,
275                     boolean_t);
276 extern int      dccp_do_unbind(conn_t *);
277 extern int      dccp_do_listen(conn_t *, struct sockaddr *, socklen_t, int,
278                     cred_t *, boolean_t);
279 extern int      dccp_do_connect(conn_t *, const struct sockaddr *, socklen_t,
280                     cred_t *, pid_t);
281 extern void     dccp_init_values(dccp_t *, dccp_t *);
282 extern void     *dccp_get_conn(void *, dccp_stack_t *);
283 extern int      dccp_set_destination(dccp_t *dccp);

285 /*
286  * Functions in dccp_bind.c
287  */
288 extern void     dccp_bind_hash_insert(dccp_df_t *, dccp_t *, int);
289 extern void     dccp_bind_hash_remove(dccp_t *);
290 extern int      dccp_bind_check(conn_t *, struct sockaddr *, socklen_t, cred_t *
291                     boolean_t);
292 extern in_port_t dccp_bindi(dccp_t *, in_port_t, const in6_addr_t *, int,
293                     boolean_t, boolean_t, boolean_t);
294 extern in_port_t dccp_update_next_port(in_port_t, const dccp_t *, boolean_t);

296 /*
297  * Functions in dccp_features.c
298  */
299 extern void     dccp_process_options(dccp_t *, dccpha_t *);
300 extern int      dccp_generate_options(dccp_t *, void **, size_t *);

302 /*
303  * Functions in dccp_stats.c
304  */
305 extern mblk_t   *dccp_snmp_get(queue_t *, mblk_t *, boolean_t);
306 extern void     *dccp_kstat_init(netstackid_t);
307 extern void     dccp_kstat_fini(netstackid_t, kstat_t *);

309 /*
310  * Functions in dccp_socket.c
311  */
312 extern sock_lower_handle_t      dccp_create(int, int, int, sock_downcalls_t **,
313                                     uint_t *, int *, int, cred_t *);
314 extern int      dccp_fallback(sock_lower_handle_t, queue_t *, boolean_t,
315                     so_proto_quiesced_cb_t, sock_quiesce_arg_t *);

317 /*
318  * Functions in dccp_input.c
319  */
320 extern void     dccp_icmp_input(void *, mblk_t *, void *, ip_recv_attr_t *);
321 extern void     dccp_input_data(void *, mblk_t *, void *, ip_recv_attr_t *);
322 extern void     dccp_rsrv(queue_t *);
323 extern void     dccp_input_listener_unbound(void *, mblk_t *, void *,
324                     ip_recv_attr_t *);
325 extern boolean_t dccp_verifyicmp(conn_t *, void *, icmph_t *, icmp6_t *,
```



```
326                     ip_recv_attr_t *);

328 /*
329  * Functions in dccp_misc.c
330  */
331 extern void     dccp_stack_cpu_add(dccp_stack_t *, processorid_t);

333 /*
334  * Functions in dccp_output.c
335  */
336 extern void     dccp_wput(queue_t *, mblk_t *);
337 extern void     dccp_wput_sock(queue_t *, mblk_t *);
338 extern void     dccp_wput_fallback(queue_t *, mblk_t *);
339 extern void     dccp_output_urgent(void *, mblk_t *, void *, ip_recv_attr_t *);
340 extern void     dccp_output(void *, mblk_t *, void *, ip_recv_attr_t *);
341 extern void     dccp_send_data(dccp_t *, mblk_t *);
342 extern void     dccp_xmit_listeners_reset(mblk_t *, ip_recv_attr_t *,
343                     ip_stack_t *, conn_t *);
344 extern void     dccp_send_synack(void *, mblk_t *, void *, ip_recv_attr_t *);
345 extern mblk_t   *dccp_xmit_mp(dccp_t *, mblk_t *, int32_t, int32_t *,
346                     mblk_t **, uint32_t, boolean_t, uint32_t *, boolean_t);
347 /* XXX following functions should be redone */
348 extern mblk_t   *dccp_generate_packet(conn_t *, mblk_t *);
349 extern mblk_t   *dccp_generate_request(conn_t *);

351 /*
352  * Functions in dccp_opt_data.c
353  */
354 extern int      dccp_opt_get(conn_t *, int, int, uchar_t *);
355 extern int      dccp_opt_set(conn_t *, uint_t, int, int, uint_t, uchar_t *,
356                     uint_t *, uchar_t *, void *, cred_t *);

358 /*
359  * Functions in dccp_tpi.c
360  */
361 extern void     dccp_do_capability_ack(dccp_t *, struct T_capability_ack *,
362                     t_uscalar_t);
363 extern void     dccp_capability_req(dccp_t *, mblk_t *);
364 extern void     dccp_err_ack(dccp_t *, mblk_t *, int, int);
365 extern void     dccp_tpi_connect(dccp_t *, mblk_t *);
366 extern int      dccp_tpi_close(queue_t *, int);
367 extern int      dccp_tpi_close_accept(queue_t *);
368 extern int      dccp_tpi_opt_get(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
369 extern int      dccp_tpi_opt_set(queue_t *, uint_t, int, int, uint_t, uchar_t *,
370                     uint_t *, uchar_t *, void *, cred_t *);
371 extern void     dccp_tpi_accept(queue_t *, mblk_t *);

373 #endif  /* _KERNEL */

375 #ifdef  __cplusplus
376 }
377 #endif

379 #endif  /* _INET_DCCP_IMPL_H */
380 #endif /* ! codereview */
```

```
**********************************************************
   14451 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_input.c
dccp: starting module template
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/ip.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"

static mblk_t   *dccp_conn_create_v4(conn_t *, conn_t *, mblk_t *,
                    ip_recv_attr_t *);
static mblk_t   *dccp_conn_create_v6(conn_t *, conn_t *, mblk_t *,
                    ip_recv_attr_t *);
static void     dccp_input_listener(void *, mblk_t *, void *, ip_recv_attr_t *);
static void     dccp_icmp_error_ipv6(dccp_t *, mblk_t *, ip_recv_attr_t *);

void
dccp_icmp_input(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
{
```



```
        conn_t  *connp = (conn_t *)arg1;
        dccp_t  *dccp = connp->conn_dccp;

        cmn_err(CE_NOTE, "dccp_input.c: dccp_icmp_input");

        /* Assume IP provides aligned packets */
        ASSERT(OK_32PTR(mp->b_rptr));
        ASSERT((MBLKL(mp) >= sizeof (ipha_t)));

        /* Verify IP version. */
        if (!(ira->ira_flags & IRAF_IS_IPV4)) {
                dccp_icmp_error_ipv6(dccp, mp, ira);
                return;
        }

}

static void
dccp_icmp_error_ipv6(dccp_t *dccp, mblk_t *mp, ip_recv_attr_t *ira)
{
        cmn_err(CE_NOTE, "dccp_input.c: dccp_icmp_error_ipv6");
}

void
dccp_rsrv(queue_t *q)
{
        cmn_err(CE_NOTE, "dccp_input.c: dccp_rsrv");
}

/*
 * Handle a REQUEST on an AF_INET6 socket; can be either IPv4 or IPv6.
 */
static mblk_t *
dccp_conn_create_v6(conn_t *lconnp, conn_t *connp, mblk_t *mp,
    ip_recv_attr_t *ira)
{
        dccp_t          *ldccp = lconnp->conn_dccp;
        dccp_t          *dccp = connp->conn_dccp;
        dccp_stack_t    *dccps = dccp->dccp_dccps;
        ipha_t          *ipha;
        ip6_t           *ip6h;
        mblk_t          *tpi_mp;
        sin6_t          sin6;
        uint_t          ifindex = ira->ira_ruifindex;

        if (ira->ira_flags & IRAF_IS_IPV4) {
                ipha = (ipha_t *)mp->b_rptr;

                connp->conn_ipversion = IPV4_VERSION;
                IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &connp->conn_laddr_v6);
                IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &connp->conn_faddr_v6);
                connp->conn_saddr_v6 = connp->conn_laddr_v6;

                sin6 = sin6_null;
                sin6.sin6_addr = connp->conn_faddr_v6;
                sin6.sin6_port = connp->conn_fport;
                sin6.sin6_family = AF_INET6;
                sin6.__sin6_src_id = ip_srcid_find_addr(&connp->conn_laddr_v6,
                    IPCL_ZONEID(lconnp), dccps->dccps_netstack);

                if (connp->conn_recv_ancillary.crb_recvdstaddr) {
                        sin6_t  sin6d;

                        sin6d = sin6_null;
                        sin6d.sin6_addr = connp->conn_laddr_v6;
                        sin6d.sin6_port = connp->conn_lport;
```

```
128                         sin6d.sin6_family = AF_INET;
129                         tpi_mp = mi_tpi_extconn_ind(NULL,
130                             (char *)&sin6d, sizeof (sin6_t),
131                             (char *)&dccp,
132                             (t_scalar_t)sizeof (intptr_t),
133                             (char *)&sin6d, sizeof (sin6_t),
134                             (t_scalar_t)ldccp->dccp_conn_req_seqnum);
135                 } else {
136                         tpi_mp = mi_tpi_conn_ind(NULL,
137                             (char *)&sin6, sizeof (sin6_t),
138                             (char *)&dccp, (t_scalar_t)sizeof (intptr_t),
139                             (t_scalar_t)ldccp->dccp_conn_req_seqnum);
140                 }
141         } else {
142                 ip6h = (ip6_t *)mp->b_rptr;

144                 connp->conn_ipversion = IPV6_VERSION;
145                 connp->conn_laddr_v6 = ip6h->ip6_dst;
146                 connp->conn_faddr_v6 = ip6h->ip6_src;
147                 connp->conn_saddr_v6 = connp->conn_laddr_v6;

149                 sin6 = sin6_null;
150                 sin6.sin6_addr = connp->conn_faddr_v6;
151                 sin6.sin6_port = connp->conn_fport;
152                 sin6.sin6_family = AF_INET6;
153                 sin6.sin6_flowinfo = ip6h->ip6_vcf & ~IPV6_VERS_AND_FLOW_MASK;
154                 sin6.__sin6_src_id = ip_srcid_find_addr(&connp->conn_laddr_v6,
155                     IPCL_ZONEID(lconnp), dccps->dccps_netstack);

157                 if (IN6_IS_ADDR_LINKSCOPE(&ip6h->ip6_src)) {
158                         /* Pass up the scope_id of remote addr */
159                         sin6.sin6_scope_id = ifindex;
160                 } else {
161                         sin6.sin6_scope_id = 0;
162                 }
163                 if (connp->conn_recv_ancillary.crb_recvdstaddr) {
164                         sin6_t  sin6d;

166                         sin6d = sin6_null;
167                         sin6.sin6_addr = connp->conn_laddr_v6;
168                         sin6d.sin6_port = connp->conn_lport;
169                         sin6d.sin6_family = AF_INET6;
170                         if (IN6_IS_ADDR_LINKSCOPE(&connp->conn_laddr_v6))
171                                 sin6d.sin6_scope_id = ifindex;

173                         tpi_mp = mi_tpi_extconn_ind(NULL,
174                             (char *)&sin6d, sizeof (sin6_t),
175                             (char *)&dccp, (t_scalar_t)sizeof (intptr_t),
176                             (char *)&sin6d, sizeof (sin6_t),
177                             (t_scalar_t)ldccp->dccp_conn_req_seqnum);
178                 } else {
179                         tpi_mp = mi_tpi_conn_ind(NULL,
180                             (char *)&sin6, sizeof (sin6_t),
181                             (char *)&dccp, (t_scalar_t)sizeof (intptr_t),
182                             (t_scalar_t)ldccp->dccp_conn_req_seqnum);
183                 }
184         }

186         /* XXX mss */
187         return (tpi_mp);
188 }

190 /*
191  * Handle a REQUEST on an AF_INET socket.
192  */
193 static mblk_t *
```



```
194 dccp_conn_create_v4(conn_t *lconnp, conn_t *connp, mblk_t *mp,
195     ip_recv_attr_t *ira)
196 {
197         dccp_t          *ldccp = lconnp->conn_dccp;
198         dccp_t          *dccp = connp->conn_dccp;
199         dccp_stack_t    *dccps = dccp->dccp_dccps;
200         ipha_t          *ipha;
201         mblk_t          *tpi_mp;
202         sin_t           sin;

204         ASSERT(ira->ira_flags & IRAF_IS_IPV4);
205         ipha = (ipha_t *)mp->b_rptr;

207         connp->conn_ipversion = IPV4_VERSION;
208         IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &connp->conn_laddr_v6);
209         IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &connp->conn_faddr_v6);
210         connp->conn_saddr_v6 = connp->conn_laddr_v6;

212         sin = sin_null;
213         sin.sin_addr.s_addr = connp->conn_faddr_v4;
214         sin.sin_port = connp->conn_fport;
215         sin.sin_family = AF_INET;

217         if (lconnp->conn_recv_ancillary.crb_recvdstaddr) {
218                 cmn_err(CE_NOTE, "ancillary");

220                 sin_t   sind;

222                 sind = sin_null;
223                 sind.sin_addr.s_addr = connp->conn_laddr_v4;
224                 sind.sin_port = connp->conn_lport;
225                 sind.sin_family = AF_INET;

227                 tpi_mp = mi_tpi_extconn_ind(NULL,
228                     (char *)&sind, sizeof (sin_t), (char *)&dccp,
229                     (t_scalar_t)sizeof (intptr_t), (char *)&sind,
230                     sizeof (sin_t), (t_scalar_t)ldccp->dccp_conn_req_seqnum);

232         } else {
233                 tpi_mp = mi_tpi_conn_ind(NULL,
234                 (char *)&sin, sizeof (sin_t),
235                 (char *)&dccp, (t_scalar_t)sizeof (intptr_t),
236                 (t_scalar_t)ldccp->dccp_conn_req_seqnum);
237         }

239         /* XXX mss */

241         return (tpi_mp);
242 }

244 static void
245 dccp_input_listener(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
246 {
247         conn_t          *lconnp = (conn_t *)arg;
248         conn_t          *econnp;
249         dccp_t          *listener = lconnp->conn_dccp;
250         dccp_t          *eager;
251         dccp_stack_t    *dccps = listener->dccp_dccps;
252         ip_stack_t      *ipst = dccps->dccps_netstack->netstack_ip;
253         dccpha_t        *dccpha;
254         squeue_t        *new_sqp;
255         mblk_t          *tpi_mp;
256         mblk_t          *mp1;
257         uint_t          ifindex = ira->ira_ruifindex;
258         uint_t          ip_hdr_len;
259         uint_t          type;
```

```
260         int             error;

262         cmn_err(CE_NOTE, "dccp_input.c: dccp_input_listener");

264         ip_hdr_len = ira->ira_ip_hdr_length;
265         dccpha = (dccpha_t *)&mp->b_rptr[ip_hdr_len];
266         type = (uint_t)dccpha->dha_type;

268         DTRACE_DCCP5(receive, mblk_t *, NULL, ip_xmit_attr_t *, lconnp->conn_ixa
269             __dtrace_dccp_void_ip_t *, mp->b_rptr, dccp_t *, listener,
270             __dtrace_dccp_dccph_t *, dccpha);

272         if (type != DCCP_PKT_REQUEST) {
273                 cmn_err(CE_NOTE, "not request pkt");

275                 /* XXX do something with a reset packet sent? */
276                 freemsg(mp);
277                 return;
278         }

280         /* XXX memory pressure */

282         /* XXX request defense */

284         /* XXX number of connections per listener */

286         ASSERT(ira->ira_sqp != NULL);
287         new_sqp = ira->ira_sqp;

289         econnp = (conn_t *)dccp_get_conn(arg2, dccps);
290         if (econnp == NULL) {
291                 cmn_err(CE_NOTE, "econnp not found (eager)");
292                 goto error2;
293         }

295         ASSERT(econnp->conn_netstack == lconnp->conn_netstack);
296         econnp->conn_sqp = new_sqp;
297         econnp->conn_initial_sqp = new_sqp;
298         econnp->conn_ixa->ixa_sqp = new_sqp;

300         econnp->conn_fport = dccpha->dha_lport;
301         econnp->conn_lport = dccpha->dha_fport;

303         error = conn_inherit_parent(lconnp, econnp);
304         if (error != 0) {
305                 cmn_err(CE_NOTE, "conn_inherit_parent failed");
306                 goto error3;
307         }

309         /* We already know the laddr of the new connection is ours */
310         econnp->conn_ixa->ixa_src_generation = ipst->ips_src_generation;

312         ASSERT(OK_32PTR(mp->b_rptr));
313         ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION ||
314             IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);

316         if (lconnp->conn_family == AF_INET) {
317                 ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION);
318                 tpi_mp = dccp_conn_create_v4(lconnp, econnp, mp, ira);
319         } else {
320                 tpi_mp = dccp_conn_create_v6(lconnp, econnp, mp, ira);
321         }

323         if (tpi_mp == NULL) {
324                 cmn_err(CE_NOTE, "tpi_mo == NULL");
325                 goto error3;
```



```
326         }

328         eager = econnp->conn_dccp;
329         SOCK_CONNID_INIT(eager->dccp_connid);

331         dccp_init_values(eager, listener);

333         ASSERT((econnp->conn_ixa->ixa_flags &
334             (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
335             IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO)) ==
336             (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
337             IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO));

339         if (!(ira->ira_flags & IRAF_IS_IPV4) && econnp->conn_bound_if == 0) {
340                 if (IN6_IS_ADDR_LINKSCOPE(&econnp->conn_faddr_v6) ||
341                     IN6_IS_ADDR_LINKSCOPE(&econnp->conn_laddr_v6)) {
342                         econnp->conn_incoming_ifindex = ifindex;
343                         econnp->conn_ixa->ixa_flags |= IXAF_SCOPEID_SET;
344                         econnp->conn_ixa->ixa_scopeid = ifindex;
345                 }
346         }

348         if (ira->ira_cred != NULL) {
349                 mblk_setcred(tpi_mp, ira->ira_cred, ira->ira_cpid);
350         }

352         if (IPCL_IS_NONSTR(lconnp)) {
353                 econnp->conn_flags |= IPCL_NONSTR;
354         }

356         /* XXX dccps is right? */
357         dccp_bind_hash_insert(&dccps->dccps_bind_fanout[
358             DCCP_BIND_HASH(econnp->conn_lport, dccps->dccps_bind_fanout_size)],

360         /* XXX CLUSTER */

362         SOCK_CONNID_BUMP(eager->dccp_connid);

364         error = dccp_set_destination(eager);
365         if (error != 0) {
366                 cmn_err(CE_NOTE, "dccp_set_destination failed.");
367                 dccp_bind_hash_remove(eager);
368                 goto error3;
369         }

371         /* Process all DCCP options */
372         dccp_process_options(eager, dccpha);

374         CONN_INC_REF(lconnp);
375         eager->dccp_conn_req_seqnum = listener->dccp_conn_req_seqnum;
376         if (++listener->dccp_conn_req_seqnum == -1) {
377                 /*
378                  * -1 is "special" and defined in TPI as something
379                  * that should never be used in T_CONN_IND
380                  */
381                 ++listener->dccp_conn_req_seqnum;
382         }

384         /* XXX SYN DEFENSE */

386         eager->dccp_state = DCCPS_REQUEST_RCVD;
387         DTRACE_DCCP6(state__change, void, NULL, ip_xmit_attr_t *,
388             econnp->conn_ixa, void, NULL, dccp_t *, eager, void, NULL,
389             int32_t, DCCPS_LISTEN);

391 /*
```

```
392         mp1 = dccp_xmit_mp(eager, eager->dccp_xmit_head, 0,
393             NULL, NULL, 0, B_FALSE, NULL, B_FALSE);
394 */
395         mp1 = dccp_generate_packet(connp, mp);
396         if (mp1 == NULL) {
397                 cmn_err(CE_NOTE, "dccp_generate_packet failed");
398                 /*
399                  * Increment the ref count as we are going to
400                  * enqueueing an mp in squeue
401                  */
402                 CONN_INC_REF(connp);
403                 goto error;
404         }

406         CONN_INC_REF(connp);

408         error = ipcl_conn_insert(connp);
409         if (error != 0) {
410                 cmn_err(CE_NOTE, "ipcl_conn_insert(connp) failed");
411                 goto error;
412         }

414         ASSERT(connp->conn_ixa->ixa_notify_cookie == connp->conn_dccp);
415         freemsg(mp);

417         /*
418          * Send the SYN-ACK. Use the right squeue so that conn_ixa is
419          * only used by one thread at a time.
420          */
421         if (connp->conn_sqp == lconnp->conn_sqp) {
422                 DTRACE_TCP5(send, mblk_t *, NULL, ip_xmit_attr_t *,
423                     connp->conn_ixa, __dtrace_dccp_void_ip_t *, mp1->b_rptr,
424                     dccp_t *, eager, __dtrace_dccp_dccph_t *,
425                     &mp1->b_rptr[connp->conn_ixa->ixa_ip_hdr_length]);
426                 (void) conn_ip_output(mp1, connp->conn_ixa);
427                 CONN_DEC_REF(connp);
428         } else {
429                 SQUEUE_ENTER_ONE(connp->conn_sqp, mp1, dccp_send_synack,
430                     connp, NULL, SQ_PROCESS, SQTAG_TCP_SEND_SYNACK); /* XXX */
431         }

433         return;
434 error:
435         freemsg(mp1);
436 error2:
437         CONN_DEC_REF(connp);
438 error3:
439         freemsg(mp);
440 }

442 void
443 dccp_input_listener_unbound(void *arg, mblk_t *mp, void *arg2,
444     ip_recv_attr_t *ira)
445 {
446         conn_t          *connp = (conn_t *)arg;
447         squeue_t        *sqp = (squeue_t *)arg2;
448         squeue_t        *new_sqp;
449         uint32_t        conn_flags;

451         cmn_err(CE_NOTE, "dccp_input.c: dccp_input_listener_unbound");

453         ASSERT(ira->ira_sqp != NULL);
454         new_sqp = ira->ira_sqp;

456         if (connp->conn_fanout == NULL) {
457                 goto done;
```



```
458         }

460         /*
461          * Bind to correct squeue.
462          */
463         if (!(connp->conn_flags & IPCL_FULLY_BOUND)) {
464                 cmn_err(CE_NOTE, "not fully bound");

466                 mutex_enter(&connp->conn_fanout->connf_lock);
467                 mutex_enter(&connp->conn_lock);

469                 if (connp->conn_ref != 4 ||
470                     connp->conn_dccp->dccp_state != DCCPS_LISTEN) {
471                         mutex_exit(&connp->conn_lock);
472                         mutex_exit(&connp->conn_fanout->connf_lock);
473                         goto done;
474                 }

476                 if (connp->conn_sqp != new_sqp) {
477                         while (connp->conn_sqp != new_sqp) {
478                                 (void) casptr(&connp->conn_sqp, sqp, new_sqp);
479                         }
480                         connp->conn_ixa->ixa_sqp = new_sqp;
481                 }

483                 do {
484                         conn_flags = connp->conn_flags;
485                         conn_flags |= IPCL_FULLY_BOUND;
486                         (void) cas32(&connp->conn_flags, connp->conn_flags,
487                             conn_flags);
488                 } while (!(connp->conn_flags & IPCL_FULLY_BOUND));

490                 mutex_exit(&connp->conn_lock);
491                 mutex_exit(&connp->conn_fanout->connf_lock);

493                 connp->conn_recv = dccp_input_listener;
494         }

496 done:
497         if (connp->conn_sqp != sqp) {
498                 CONN_INC_REF(connp);
499                 SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_recv, connp,
500                     ira, SQ_FILL, SQTAG_DCCP_CONN_REQ_UNBOUND);
501         } else {
502                 dccp_input_listener(connp, mp, sqp, ira);
503         }
504 }

506 boolean_t
507 dccp_verifyicmp(conn_t *connp, void *arg2, icmph_t *icmph, icmp6_t *icmp6,
508     ip_recv_attr_t *ira)
509 {
510         cmn_err(CE_NOTE, "dccp_input.c: dccp_verifyicmp");

512         return (B_TRUE);
513 }

515 /*
516  * After a request-response-ack all packets end up here.
517  */
518 void
519 dccp_input_data(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
520 {
521         conn_t          *connp = (conn_t *)arg;
522         squeue_t        *sqp = (squeue_t *)arg2;
523         dccp_t          *dccp = connp->conn_dccp;
```

```
524         dccp_stack_t    *dccps = dccp->dccp_dccps;
525         dccpha_t        *dccpha;
526         uchar_t         *iphdr;
527         uchar_t         *rptr;
528         sock_upcalls_t  *sockupcalls;
529         uint_t          ip_hdr_len;

531         cmn_err(CE_NOTE, "dccp_input.c: dccp_input_data");

533         iphdr = mp->b_rptr;
534         rptr = mp->b_rptr;
535         ASSERT(OK_32PTR(rptr));

537         ip_hdr_len = ira->ira_ip_hdr_length;

539         ASSERT(DB_TYPE(mp) == M_DATA);
540         ASSERT(mp->b_next == NULL);

542         dccpha = (dccpha_t *)&rptr[ip_hdr_len];
543 }

545 #endif /* ! codereview */
```

```
**********************************************************
     807 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_ip.h
dccp: starting module template
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner. All rights reserved.
 */

#ifndef _INET_DCCP_DCCP_IP_H
#define _INET_DCCP_DCCP_IP_H

#include <netinet/dccp.h>
#include <inet/dccp/dccp_stack.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * DCCP functions for IP
 */
extern void     dccp_ddi_g_init(void);
extern void     dccp_ddi_g_destroy(void);


#ifdef  __cplusplus
}
#endif

#endif  /* _INET_DCCP_DCCP_IP_H */
#endif /* ! codereview */
```

```
**********************************************************
    1505 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_misc.c
dccp: add dccp_misc.c
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * Functions related to XXX.
 */

#include <sys/types.h>
#include <inet/common.h>

#include "dccp_impl.h"

/*
 * When a CPU is added, we need to allocate the per CPU stats struct.
 */
void
dccp_stack_cpu_add(dccp_stack_t *dccps, processorid_t cpu_seqid)
{
        int     i;

        if (cpu_seqid < dccps->dccps_sc_cnt) {
                return;
        }

        for (i = dccps->dccps_sc_cnt; i <= cpu_seqid; i++) {
                ASSERT(dccps->dccps_sc[i] == NULL);
                dccps->dccps_sc[i] = kmem_zalloc(sizeof (dccp_stats_cpu_t),
                    KM_SLEEP);
        }

        membar_producer();
        dccps->dccps_sc_cnt = cpu_seqid + 1;
}
#endif /* ! codereview */
```

```
**********************************************************
    3415 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_opt_data.c
dccp: starting with options
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * This file contains functions related to getting and setting options
 * thought the getsockopt and setsockopt socket functions.
 */

#include <sys/types.h>
#include <sys/stream.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/xti_xtiopt.h>
#include <sys/xti_inet.h>
#include <sys/policy.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/optcom.h>
#include <netinet/ip.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"

static int dccp_opt_default(queue_t *, int, int, uchar_t *);

/*
 * Supported options.
 */
opdes_t dccp_opt_arr[] = {
{ SO_DEBUG, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
};
```



```
/*
 * Supported levels.
 */
optlevel_t dccp_valid_levels_arr[] = {
        SOL_SOCKET,
};

#define DCCP_OPT_ARR_CNT        A_CNT(dccp_opt_arr)
#define DCCP_VALID_LEVELS_CNT   A_CNT(dccp_valid_levels_arr)

uint_t dccp_max_optsize;

/*
 * Options database object.
 */
optdb_obj_t dccp_opt_obj = {
        dccp_opt_default,
        dccp_tpi_opt_get,
        dccp_tpi_opt_set,
        DCCP_OPT_ARR_CNT,
        dccp_opt_arr,
        DCCP_VALID_LEVELS_CNT,
        dccp_valid_levels_arr,
};

/*
 * Default value for certain options.
 */
int
dccp_opt_default(queue_t *q, int level, int name, uchar_t *ptr)
{
        dccp_stack_t    *dccps = Q_TO_DCCP(q)->dccp_dccps;
        int32_t         *il = (int32_t *)ptr;

        return (sizeof (int));
}

int
dccp_opt_get(conn_t *connp, int level, int name, uchar_t *ptr)
{
        dccp_t          *dccp = connp->conn_dccp;
        conn_opt_arg_t  coas;
        int             retval;

        coas.coa_connp = connp;
        coas.coa_ixa = connp->conn_ixa;
        coas.coa_ipp = &connp->conn_xmit_ipp;
        coas.coa_ancillary = B_FALSE;
        coas.coa_changed = 0;

        switch (level) {
        case SOL_SOCKET:
                break;
        case IPPROTO_TCP:
                break;
        case IPPROTO_IP:
                break;
        case IPPROTO_IPV6:
                break;
        }

        mutex_enter(&connp->conn_lock);
        retval = conn_opt_get(&coas, level, name, ptr);
        mutex_exit(&connp->conn_lock);

        return (retval);
```

```
128 }

130 /* ARGSUSED */
131 int
132 dccp_opt_set(conn_t *connp, uint_t optset_context, int level, int name,
133     uint_t inlen, uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
134     void *thisdg_attrs, cred_t *cr)
135 {
136         dccp_t          *dccp = connp->conn_dccp;
137         dccp_stack_t    *dccps = dccp->dccp_dccps;
138         conn_opt_arg_t  coas;
139         int             *i1 = (int *)invalp;
140         int             error;

142         coas.coa_connp = connp;
143         coas.coa_ancillary = B_FALSE;
144         coas.coa_changed = 0;

146         error = conn_opt_set(&coas, level, name, inlen, invalp,
147             B_FALSE, cr);
148         if (error !=0) {
149                 *outlenp = 0;
150                 return (error);
151         }

153         return (0);
154 }
155 #endif /* ! codereview */
```

```
**********************************************************
   14126 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_output.c
dccp: clean up
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * Functions related to the output path.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ipsec_impl.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"
#include "dccp_stack.h"

static void     dccp_xmit_early_reset(char *, mblk_t *, uint32_t, uint32_t,
                    int, ip_recv_attr_t *, ip_stack_t *, conn_t *);
static boolean_t        dccp_send_rst_chk(dccp_stack_t *);
```



```
/*
 * STREAMS
 */
void
dccp_wput(queue_t *q, mblk_t *mp)
{
        cmn_err(CE_NOTE, "dccp_output.c: dccp_wput\n");
}

/*
 *
 */
void
dccp_wput_sock(queue_t *wq, mblk_t *mp)
{
        conn_t  *connp = Q_TO_CONN(wq);
        dccp_t  *dccp = connp->conn_dccp;
        struct T_capability_req *car = (struct T_capability_req *)mp->b_rptr;

        ASSERT(wq->q_qinfo == &dccp_sock_winit);
        wq->q_qinfo = &dccp_winit;

        ASSERT(IPCL_IS_TCP(connp));
        ASSERT(DCCP_IS_SOCKET(dccp));

        if (DB_TYPE(mp) == M_PCPROTO &&
            MBLKL(mp) == sizeof (struct T_capability_req) &&
            car->PRIM_type == T_CAPABILITY_REQ) {
                dccp_capability_req(dccp, mp);
                return;
        }

        dccp_wput(wq, mp);
}

/* ARGSUSED */
void
dccp_wput_fallback(queue_t *eq, mblk_t *mp)
{
        cmn_err(CE_NOTE, "dccp_output.c: dccp_wput_fallback");

#ifdef DEBUG
        cmn_err(CE_CONT, "tcp_wput_fallback: Message during fallback \n");
#endif /* DEBUG */

        freemsg(mp);
}

void
dccp_output(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *dummy)
{
        cmn_err(CE_NOTE, "dccp_output.c: dccp_output");
}

void
dccp_output_urgent(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *dummy)
{
        cmn_err(CE_NOTE, "dccp_output.c: dccp_output_urgent");
}

// XXX */

#pragma inline(dccp_send_data)

void
dccp_send_data(dccp_t *dccp, mblk_t *mp)
```

```
128 {
129         conn_t  *connp = dccp->dccp_connp;

131         /* XXX dtrace */

133         ASSERT(connp->conn_ixa->ixa_notify_cookie == connp->conn_tcp);
134         (void) conn_ip_output(mp, connp->conn_ixa);
135 }

137 /*
138  * Send a reset as response to an incoming packet or
139  * reset a connection.
140  */
141 void
142 dccp_xmit_listeners_reset(mblk_t *mp, ip_recv_attr_t *ira, ip_stack_t *ipst,
143     conn_t *connp)
144 {
145         netstack_t      *ns = ipst->ips_netstack;
146         dccp_stack_t    *dccps = ns->netstack_dccp;
147         ipsec_stack_t   *ipss = dccps->dccps_netstack->netstack_ipsec;
148         dccpha_t        *dccpha;
149         ipha_t          *ipha;
150         ip6_t           *ip6h;
151         uchar_t         *rptr;
152         uint32_t        seq_len;
153         uint_t          ip_hdr_len = ira->ira_ip_hdr_length;
154         boolean_t       policy_present;

156         cmn_err(CE_NOTE, "dccp_output.c: dccp_xmit_listeners_reset");

158         DCCP_STAT(dccps, dccp_no_listener);

160         if (IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION) {
161                 policy_present = ipss->ipsec_inbound_v4_policy_present;
162                 ipha = (ipha_t *)mp->b_rptr;
163                 ip6h = NULL;
164         } else {
165                 policy_present = ipss->ipsec_inbound_v6_policy_present;
166                 ipha = NULL;
167                 ip6h = (ip6_t *)mp->b_rptr;
168         }

170         if (policy_present) {
171                 mp = ipsec_check_global_policy(mp, (conn_t *)NULL, ipha, ip6h,
172                     ira, ns);
173                 if (mp == NULL) {
174                         return;
175                 }
176         }

178         rptr = mp->b_rptr;

180         dccpha = (dccpha_t *)&rptr[ip_hdr_len];

182         seq_len = msgdsize(mp) - (ip_hdr_len);

184         dccp_xmit_early_reset("no dccp, reset", mp, 0,
185             0, 0, ira, ipst, connp);
186 }

188 /*
189  * RFC 4340, Section 8.1.3
190  */
191 static void
192 dccp_xmit_early_reset(char *str, mblk_t *mp, uint32_t seq, uint32_t ack, int ctl
193     ip_recv_attr_t *ira, ip_stack_t *ipst, conn_t *connp)
```



```
194 {
195         dccpha_t        *dccpha;
196         dccpha_t        *nmp_dccpha;
197         dccpha_ack_t    *nmp_dccpha_ack;
198         dccpha_reset_t  *dccpha_reset;
199         dccpha_reset_t  *nmp_dccpha_reset;
200         dccpha_ext_t    *dccpha_ext;
201         dccpha_ext_t    *nmp_dccpha_ext;
202         netstack_t      *ns = ipst->ips_netstack;
203         dccp_stack_t    *dccps = ns->netstack_dccp;
204         ip6_t           *ip6h;
205         ipha_t          *ipha;
206         ipha_t          *nmp_ipha;
207         ip_xmit_attr_t  ixas;
208         ip_xmit_attr_t  *ixa;
209         in6_addr_t      v6addr;
210         ipaddr_t        v4addr;
211         mblk_t          *nmp;
212         uint64_t        pkt_ack;
213         uint_t          ip_hdr_len = ira->ira_ip_hdr_length;
214         ushort_t        port;
215         ushort_t        len;

217         cmn_err(CE_NOTE, "dccp_output.c: dccp_xmit_early_reset");

219         if (!dccp_send_rst_chk(dccps)) {
220                 cmn_err(CE_NOTE, "dccp_output.c: not sending reset packet");
221                 DCCP_STAT(dccps, dccp_rst_unsent);
222                 freemsg(mp);
223                 return;
224         }

226                 bzero(&ixas, sizeof (ixas));
227                 ixa = &ixas;

229                 ixa->ixa_flags |= IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE;
230                 ixa->ixa_protocol = IPPROTO_DCCP;
231                 ixa->ixa_zoneid = ira->ira_zoneid;
232                 ixa->ixa_ifindex = 0;
233                 ixa->ixa_ipst = ipst;
234                 ixa->ixa_cred = kcred;
235                 ixa->ixa_cpid = NOPID;

237         if (str && dccps->dccps_dbg) {
238                 (void) strlog(DCCP_MOD_ID, 0, 1, SL_TRACE,
239                     "dccp_xmit_early_reset: '%s', seq 0x%x, ack 0x%x, "
240                     "flags 0x%x",
241                     str, seq, ack, ctl);
242         }

244         if (IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION) {
245                 ipha = (ipha_t *)mp->b_rptr;

247         } else {
248                 /* XXX */
249         }

251         /*
252          * Allocate a new DCCP reset message
253          */
254         len = ip_hdr_len + sizeof (dccpha_t) + sizeof (dccpha_ext_t) + sizeof (d
255         nmp = allocb(len, BPRI_MED);
256         if (nmp == NULL) {
257                 cmn_err(CE_NOTE, "alloc failed");
258                 return;
259         }
```

```
260         bcopy(mp->b_rptr, nmp->b_wptr, ip_hdr_len + sizeof (dccpha_t));

262         nmp_dccpha = (dccpha_t *)&nmp->b_rptr[ip_hdr_len];
263         nmp_dccpha->dha_offset = 7;

265         if (IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION) {
266                 nmp_ipha = (ipha_t *)nmp->b_rptr;

268                 nmp_ipha->ipha_length = htons(len);
269                 nmp_ipha->ipha_src = ipha->ipha_dst;
270                 nmp_ipha->ipha_dst = ipha->ipha_src;

272                 ixa->ixa_flags |= IXAF_IS_IPV4;
273                 ixa->ixa_ip_hdr_length = ip_hdr_len;
274         } else {
275                 cmn_err(CE_NOTE, "not v4");
276         }

278         dccpha = (dccpha_t *)&mp->b_rptr[ip_hdr_len];

280         nmp->b_wptr = &nmp->b_rptr[len];

282         ixa->ixa_pktlen = len; // ?

284         nmp_dccpha->dha_fport = dccpha->dha_lport;
285         nmp_dccpha->dha_lport = dccpha->dha_fport;
286         nmp_dccpha->dha_type = DCCP_PKT_RESET;
287         nmp_dccpha->dha_x = 1;
288         nmp_dccpha->dha_res_seq = 0;
289         nmp_dccpha->dha_seq = 0;

291         nmp_dccpha->dha_sum = htons(sizeof (dccpha_t) + sizeof (dccpha_ext_t) +

293         dccpha_ext = (dccpha_ext_t *)&mp->b_rptr[ip_hdr_len + sizeof (dccpha_t)]
294         nmp_dccpha_ext = (dccpha_ext_t *)&nmp->b_rptr[ip_hdr_len + sizeof (dccph
295         nmp_dccpha_ext->dha_ext_seq = 0;

297         len = ip_hdr_len + sizeof (dccpha_t) + sizeof (dccpha_ext_t);
298         nmp_dccpha_ack = (dccpha_ack_t *)&nmp->b_rptr[len];
299         nmp_dccpha_ack->dha_ack_high = dccpha->dha_seq;
300         nmp_dccpha_ack->dha_ack_low = dccpha_ext->dha_ext_seq;

302         len = ip_hdr_len + sizeof (dccpha_t) + sizeof (dccpha_ext_t) + sizeof (d
303         nmp_dccpha_reset = (dccpha_reset_t *)&nmp->b_rptr[len];
304         nmp_dccpha_reset->dha_reset_code = 7;
305         nmp_dccpha_reset->dha_reset_data[0] = 0;
306         nmp_dccpha_reset->dha_reset_data[1] = 0;
307         nmp_dccpha_reset->dha_reset_data[2] = 0;

309         (void) ip_output_simple(nmp, ixa);

311         ixa_cleanup(ixa);
312 }

314 /*
315  *
316  */
317 static boolean_t
318 dccp_send_rst_chk(dccp_stack_t *dccps)
319 {
320         int64_t now;

322         if (dccps->dccps_rst_sent_rate_enabled != 0) {
323                 now = ddi_get_lbolt64();
324                 if (TICK_TO_MSEC(now - dccps->dccps_last_rst_intrvl) >
325                     1 * SECONDS) {
```



```
326                         dccps->dccps_last_rst_intrvl = now;
327                         dccps->dccps_rst_cnt = 1;
328                 } else if (++dccps->dccps_rst_cnt > dccps->dccps_rst_sent_rate)
329                         return (B_FALSE);
330                 }
331         }

333         return (B_TRUE);
334 }

336 void
337 dccp_send_synack(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *dummy)
338 {
339         cmn_err(CE_NOTE, "dccp_output.c: dccp_send_synack");
340 }

342 mblk_t *
343 dccp_xmit_mp(dccp_t *dccp, mblk_t *mp, int32_t max_to_send, int32_t *offset,
344     mblk_t **end_mp, uint32_t seq, boolean_t sendall, uint32_t *seg_len,
345     boolean_t rexmit)
346 {
347         conn_t          *connp = dccp->dccp_connp;
348         dccp_stack_t    *dccps = dccp->dccp_dccps;
349         dccpha_t        *dccpha;
350         dccpha_ext_t    *dccpha_ext;
351         dccpha_ack_t    *dccpha_ack;
352         dccpha_srv_t    *dccpha_srv;
353         ip_xmit_attr_t  *ixa = connp->conn_ixa;
354         mblk_t          *mp1;
355         uchar_t         *rptr;
356         ushort_t        len;
357         int             data_length;

359         cmn_err(CE_NOTE, "dccp_output.c: dccp_xmit_mp");

361         // dccpha_t already in iphc_len?
362         len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) + sizeof (dccpha_a

364         //mp1 = dccp_generate_packet(connp, mp);
365         mp1 = allocb(len, BPRI_MED);
366         if (mp1 == NULL) {
367                 cmn_err(CE_NOTE, "allocb failed");
368                 return (NULL);
369         }

371         data_length = 0;

373         rptr = mp1->b_rptr;
374         mp1->b_wptr = &mp1->b_rptr[len];
375         bcopy(connp->conn_ht_iphc, rptr, connp->conn_ht_iphc_len);
376         dccpha = (dccpha_t *)&rptr[ixa->ixa_ip_hdr_length];
377         dccpha->dha_type = DCCP_PKT_RESPONSE;
378         dccpha->dha_offset = 8;
379         dccpha->dha_x = 1;
380         dccpha->dha_ccval = 0;
381         dccpha->dha_cscov = 0;
382         dccpha->dha_reserved = 0;
383         dccpha->dha_res_seq = 0;
384         dccpha->dha_seq = 0;

386         dccpha_ext = (dccpha_ext_t *)&rptr[ixa->ixa_ip_hdr_length + sizeof (dccp
387         dccpha_ext->dha_ext_seq = 0;

389         dccpha_ack = (dccpha_ack_t *)&rptr[ixa->ixa_ip_hdr_length + sizeof (dccp
390         dccpha_ack->dha_ack_reserved = 0;
391         dccpha_ack->dha_ack_high = 0;
```

```
392         dccpha_ack->dha_ack_low = 0;

394         dccpha_srv = (dccpha_srv_t *)&rptr[ixa->ixa_ip_hdr_length + sizeof (dccp
395         dccpha_srv->dha_srv_code = 0;

397         return (mp1);
398 }

400 mblk_t *
401 dccp_generate_packet(conn_t *connp, mblk_t *mp)
402 {
403         dccpha_t        *dccpha;
404         dccpha_ext_t    *dccpha_ext;
405         dccpha_ack_t    *dccpha_ack;
406         dccpha_srv_t    *dccpha_srv;
407         mblk_t          *mp1;
408         uint16_t        ack_high;
409         uint32_t        ack_low;
410 //      uint_t          ip_hdr_len = ira->ira_ip_hdr_length;
411         ip_xmit_attr_t  *ixa = connp->conn_ixa;
412         uint_t          ip_hdr_len;
413         uint_t          len;
414         uint_t          total_hdr_len;
415         uchar_t         *rptr;
416         dccp_t          *dccp = connp->conn_dccp;
417         void            *options;
418         size_t          opt_len;
419         int             error;

421         cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_packet");

423         ip_hdr_len = ixa->ixa_ip_hdr_length;

425         if (mp == NULL) {
426                 cmn_err(CE_NOTE, "NULL pointer mp");
427                 return (NULL);
428         }

430         dccpha = (dccpha_t *)&mp->b_rptr[ip_hdr_len];
431         dccpha_ext = (dccpha_ext_t *)&mp->b_rptr[ip_hdr_len + sizeof (dccpha_t)]

433         ack_high = dccpha->dha_seq;
434         ack_low = dccpha_ext->dha_ext_seq;

436         error = dccp_generate_options(dccp, &options, &opt_len);
437         if (error != 0) {
438                 cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_options failed");
439         }
440         cmn_err(CE_NOTE, "generated options len: %d", (int) opt_len);


443         /*
444          * conn_ht_iphc_len = ip_hdr_length (20) + ulp_hdr_length
445          * (20) simple ip header (without vtag or options)
446          */
447         total_hdr_len = len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) +
448         mp1 = allocb(len, BPRI_MED);
449         if (mp1 == NULL) {
450                 cmn_err(CE_NOTE, "allocb failed");
451                 return (NULL);
452         }

454         rptr = mp1->b_rptr;
455         mp1->b_wptr = &mp1->b_rptr[len];

457         bcopy(options, &mp1->b_rptr[len-opt_len], opt_len);
```



```
458         bcopy(connp->conn_ht_iphc, rptr, connp->conn_ht_iphc_len);
459         dccpha = (dccpha_t *)&rptr[ip_hdr_len];

461         dccpha->dha_type = DCCP_PKT_RESPONSE;
462         dccpha->dha_offset = 7 + (opt_len / 4);
463         dccpha->dha_x = 1;
464         dccpha->dha_ccval = 0;
465         dccpha->dha_cscov = 0;
466         dccpha->dha_reserved = 0;
467         dccpha->dha_res_seq = 0;
468         dccpha->dha_seq = 0;
469         dccpha->dha_sum = htons(sizeof (dccpha_t) + sizeof (dccpha_ext_t) + size


472         dccpha_ext = (dccpha_ext_t *)&mp1->b_rptr[ip_hdr_len + sizeof (dccpha_t)
473         dccpha_ext->dha_ext_seq = 1234567;

475         dccpha_ack = (dccpha_ack_t *)&mp1->b_rptr[ip_hdr_len + sizeof (dccpha_t)
476         dccpha_ack->dha_ack_high = ack_high;
477         dccpha_ack->dha_ack_low = ack_low;

479         dccpha_srv = (dccpha_srv_t *)&mp1->b_rptr[ip_hdr_len + sizeof (dccpha_t)
480         dccpha_srv->dha_srv_code = 0;

482         ixa->ixa_pktlen = total_hdr_len;

484         if (ixa->ixa_flags & IXAF_IS_IPV4) {
485                 ((ipha_t *)rptr)->ipha_length = htons(total_hdr_len);
486         } else {
487                 ip6_t *ip6 = (ip6_t *)rptr;

489                 ip6->ip6_plen = htons(total_hdr_len - IPV6_HDR_LEN);
490         }

492         cmn_err(CE_NOTE, "IPHC LEN: %d", connp->conn_ht_iphc_len);
493         cmn_err(CE_NOTE, "TOTAL LEN: %d", total_hdr_len);

495         kmem_free(options, opt_len);

497         return (mp1);
498 }

500 /*
501  * Generate a request packet. Must use 48-bit sequence
502  * numbers.
503  */
504 mblk_t *
505 dccp_generate_request(conn_t *connp)
506 {
507         dccp_t          *dccp = connp->conn_dccp;
508         dccpha_t        *dccpha;
509         dccpha_ext_t    *dccpha_ext;
510         dccpha_srv_t    *dccpha_srv;
511         ip_xmit_attr_t  *ixa = connp->conn_ixa;
512         mblk_t          *mp;
513         uchar_t         *rptr;
514         uint_t          total_hdr_len;
515         uint_t          len = ixa->ixa_ip_hdr_length;

517         cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_request");

519         total_hdr_len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) + sizeof
520         mp = allocb(total_hdr_len, BPRI_MED);
521         if (mp == NULL) {
522                 return (NULL);
523         }
```

```
525         rptr = mp->b_rptr;
526         /* Copy in the template header */
527         bcopy(connp->conn_ht_iphc, rptr, connp->conn_ht_iphc_len);

529         dccpha = (dccpha_t *)&rptr[ixa->ixa_ip_hdr_length];
530         dccpha->dha_type = DCCP_PKT_REQUEST;
531         dccpha->dha_x = 1;

533         /* Extended sequence number */
534         len += sizeof (dccpha_t);
535         dccpha_ext = (dccpha_ext_t *)&rptr[len];

537         /* Service number */
538         len += sizeof (dccpha_ext_t);
539         dccpha_srv = (dccpha_srv_t *)&rptr[len];
540         dccpha_srv->dha_srv_code = 0;

542         ixa->ixa_pktlen = total_hdr_len;
543         if (ixa->ixa_flags & IXAF_IS_IPV4) {
544                 ((ipha_t *)rptr)->ipha_length = htons(total_hdr_len);
545         } else {
546                 ip6_t *ip6 = (ip6_t *)rptr;

548                 ip6->ip6_plen = htons(total_hdr_len - IPV6_HDR_LEN);
549         }

551         return (mp);
552 }
553 #endif /* ! codereview */
```

```
**********************************************************
   15084 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_socket.c
dccp: starting module template
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * This file contains function related to the socket interface.
 */

#include <sys/types.h>
#include <sys/strlog.h>
#include <sys/policy.h>
#include <sys/sockio.h>
#include <sys/strsubr.h>
#include <sys/strsun.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/socketvar.h>

#include <inet/common.h>
#include <inet/proto_set.h>
#include <inet/ip.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"
#include "dccp_stack.h"

static void     dccp_activate(sock_lower_handle_t, sock_upper_handle_t,
                    sock_upcalls_t *, int, cred_t *);
static int      dccp_accept(sock_lower_handle_t, sock_lower_handle_t,
                    sock_upper_handle_t, cred_t *);
static int      dccp_bind(sock_lower_handle_t, struct sockaddr *,
                    socklen_t, cred_t *);
```



```
static int      dccp_listen(sock_lower_handle_t, int, cred_t *);
static int      dccp_connect(sock_lower_handle_t, const struct sockaddr *,
                    socklen_t, sock_connid_t *, cred_t *);
static int      dccp_getpeername(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
static int      dccp_getsockname(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
static int      dccp_getsockopt(sock_lower_handle_t, int, int, void *,
                    socklen_t *, cred_t *);
static int      dccp_setsockopt(sock_lower_handle_t, int, int, const void *,
                    socklen_t, cred_t *);
static int      dccp_send(sock_lower_handle_t, mblk_t *, struct nmsghdr *,
                    cred_t *);
static int      dccp_shutdown(sock_lower_handle_t, int, cred_t *);
static void     dccp_clr_flowctrl(sock_lower_handle_t);
static int      dccp_ioctl(sock_lower_handle_t, int, intptr_t, int, int32_t *,
                    cred_t *);
static int      dccp_close(sock_lower_handle_t, int, cred_t *);

sock_downcalls_t sock_dccp_downcalls = {
        dccp_activate,          /* sd_activate */
        dccp_accept,            /* sd_accept */
        dccp_bind,              /* sd_bind */
        dccp_listen,            /* sd_listen */
        dccp_connect,           /* sd_connect */
        dccp_getpeername,       /* sd_getpeername */
        dccp_getsockname,       /* sd_getsockname */
        dccp_getsockopt,        /* sd_getsockopt */
        dccp_setsockopt,        /* sd_setsockopt */
        dccp_send,              /* sd_send */
        NULL,                   /* sd_send_uio */
        NULL,                   /* sd_recv_uio */
        NULL,                   /* sd_poll */
        dccp_shutdown,          /* sd_shutdown */
        dccp_clr_flowctrl,      /* sd_setflowctrl */
        dccp_ioctl,             /* sd_ioctl */
        dccp_close,             /* sd_close */
};

/* ARGSUSED */
static void
dccp_activate(sock_lower_handle_t proto_handle, sock_upper_handle_t sock_handle,
    sock_upcalls_t *sock_upcalls, int flags, cred_t *cr)
{
        conn_t  *connp = (conn_t *)proto_handle;
        struct sock_proto_props         sopp;
        //extern struct module_info     dccp_rinfo;

        cmn_err(CE_NOTE, "dccp_socket.c: dccp_activate");

        ASSERT(connp->conn_upper_handle == NULL);

        /* All Solaris components should pass a cred for this operation */
        ASSERT(cr != NULL);

        sopp.sopp_flags = SOCKOPT_RCVHIWAT | SOCKOPT_RCVLOWAT |
            SOCKOPT_MAXPSZ | SOCKOPT_MAXBLK | SOCKOPT_RCVTIMER |
            SOCKOPT_RCVTHRESH | SOCKOPT_MAXADDRLEN | SOCKOPT_MINPSZ;

        sopp.sopp_rxhiwat = SOCKET_RECVHIWATER;
        sopp.sopp_rxlowat = SOCKET_RECVLOWATER;
        sopp.sopp_maxpsz = INFPSZ;
        sopp.sopp_maxblk = INFPSZ;
        sopp.sopp_rcvtimer = SOCKET_TIMER_INTERVAL;
        sopp.sopp_rcvthresh = SOCKET_RECVHIWATER >> 3;
        sopp.sopp_maxaddrlen = sizeof (sin6_t);
```

```
128 /*
129         sopp.sopp_minpsz = (dccp_rinfo.mi_minpsz == 1) ? 0 :
130             dccp_rinfo.mi_minpsz;
131 */
132         connp->conn_upcalls = sock_upcalls;
133         connp->conn_upper_handle = sock_handle;

135         /* XXX */
136         (*connp->conn_upcalls->su_set_proto_props)(connp->conn_upper_handle,
137             &sopp);
138 }

140 /*ARGSUSED*/
141 static int
142 dccp_accept(sock_lower_handle_t lproto_handle,
143     sock_lower_handle_t eproto_handle, sock_upper_handle_t sock_handle,
144     cred_t *cr)
145 {
146         conn_t  *lconnp, *econnp;
147         dccp_t  *listener, *eager;

149         cmn_err(CE_NOTE, "dccp_socket.c: dccp_accept");

151         econnp = (conn_t *)eproto_handle;
152         eager = econnp->conn_dccp;

154         ASSERT(IPCL_IS_NONSTR(econnp));

156         ASSERT(econnp->conn_upper_handle == NULL ||
157             econnp->conn_upper_handle == sock_handle);

159         return (ENOTSUP);
160 }

162 static int
163 dccp_bind(sock_lower_handle_t proto_handle, struct sockaddr *sa,
164     socklen_t len, cred_t *cr)
165 {
166         conn_t  *connp = (conn_t *)proto_handle;
167         int     error;

169         cmn_err(CE_NOTE, "dccp_socket.c: dccp_bind");

171         ASSERT(connp->conn_upper_handle != NULL);

173         /* All Solaris components should pass a cred for this operation */
174         ASSERT(cr != NULL);

176         error = squeue_synch_enter(connp, NULL);
177         if (error != 0) {
178                 /* Failed to enter */
179                 return (ENOSR);
180         }

182         /* Binding to NULL address means unbind */
183         if (sa == NULL) {
184                 if (connp->conn_dccp->dccp_state < DCCPS_LISTEN) {
185                         error = dccp_do_unbind(connp);
186                 } else {
187                         error = EINVAL;
188                 }
189         } else {
190                 error = dccp_do_bind(connp, sa, len, cr, B_TRUE);
191         }

193         squeue_synch_exit(connp);
```



```
195         if (error < 0) {
196                 if (error == -TOUTSTATE) {
197                         error = EINVAL;
198                 } else {
199                         error = proto_tlitosyserr(-error);
200                 }
201         }

203         return (error);
204 }

206 /* ARGSUSED */
207 static int
208 dccp_listen(sock_lower_handle_t proto_handle, int backlog, cred_t *cr)
209 {
210         conn_t  *connp = (conn_t *)proto_handle;
211         dccp_t  *dccp = connp->conn_dccp;
212         int     error;

214         cmn_err(CE_NOTE, "dccp_socket.c: dccp_listen");

216         ASSERT(connp->conn_upper_handle != NULL);

218         /* All Solaris components should pass a cred for this operation */
219         ASSERT(cr != NULL);

221         error = squeue_synch_enter(connp, NULL);
222         if (error != 0) {
223                 /* Failed to enter */
224                 return (ENOBUFS);
225         }

227         error = dccp_do_listen(connp, NULL, 0, backlog, cr, B_FALSE);
228         if (error == 0) {
229                 /* XXX:DCCP */
230                 (*connp->conn_upcalls->su_opctl)(connp->conn_upper_handle,
231                     SOCK_OPCTL_ENAB_ACCEPT,
232                     (uintptr_t)(10));
233         } else if (error < 0) {
234                 if (error == -TOUTSTATE) {
235                         error = EINVAL;
236                 } else {
237                         error = proto_tlitosyserr(-error);
238                 }
239         }

241         squeue_synch_exit(connp);

243         return (error);
244 }

246 static int
247 dccp_connect(sock_lower_handle_t proto_handle, const struct sockaddr *sa,
248     socklen_t len, sock_connid_t *id, cred_t *cr)
249 {
250         conn_t  *connp = (conn_t *)proto_handle;
251         int     error;

253         cmn_err(CE_NOTE, "dccp_socket.c: dccp_connect");

255         ASSERT(connp->conn_upper_handle != NULL);

257         /* All Solaris components should pass a cred for this operation */
258         ASSERT(cr != NULL);
```

```
260         error = proto_verify_ip_addr(connp->conn_family, sa, len);
261         if (error != 0) {
262                 return (error);
263         }

265         error = squeue_synch_enter(connp, NULL);
266         if (error != 0) {
267                 /* Failed to enter */
268                 return (ENOSR);
269         }

271         error = dccp_do_connect(connp, sa, len, cr, curproc->p_pid);
272         if (error == 0) {
273                 *id = connp->conn_dccp->dccp_connid;
274         } else if (error < 0) {
275                 if (error == -TOUTSTATE) {
276                         switch (connp->conn_dccp->dccp_state) {
277                         case DCCPS_ACK_SENT:
278                                 error = EALREADY;
279                                 break;
280                         case DCCPS_ESTABLISHED:
281                                 error = EISCONN;
282                                 break;
283                         case DCCPS_LISTEN:
284                                 error = EOPNOTSUPP;
285                                 break;
286                         default:
287                                 error = EINVAL;
288                                 break;
289                         }
290                 } else {
291                         error = proto_tlitosyserr(-error);
292                 }
293         }

295         squeue_synch_exit(connp);

297         cmn_err(CE_NOTE, "dccp_connect.c: exit %d", error);
298         return ((error == 0) ? EINPROGRESS : error);
299 }

301 /* ARGSUSED3 */
302 static int
303 dccp_getpeername(sock_lower_handle_t proto_handle, struct sockaddr *addr,
304     socklen_t *addrlenp, cred_t *cr)
305 {
306         conn_t  *connp = (conn_t *)proto_handle;
307         dccp_t  *dccp = connp->conn_dccp;

309         cmn_err(CE_NOTE, "dccp_socket.c: dccp_getpeername");

311         /* All Solaris components should pass a cred for this operation */
312         ASSERT(cr != NULL);

314         ASSERT(dccp != NULL);
315         if (dccp->dccp_state < DCCPS_ACK_RCVD) {
316                 return (ENOTCONN);
317         }

319         return (conn_getpeername(connp, addr, addrlenp));
320 }

322 /* ARGSUSED3 */
323 static int
324 dccp_getsockname(sock_lower_handle_t proto_handle, struct sockaddr *addr,
325     socklen_t *addrlenp, cred_t *cr)
```



```
326 {
327         conn_t  *connp = (conn_t *)proto_handle;
328         int     error;

330         cmn_err(CE_NOTE, "dccp_socket.c: dccp_getsockname");

332         /* All Solaris components should pass a cred for this operation */
333         ASSERT(cr != NULL);

335         /* XXX UDP has locks here, TCP not */
336         mutex_enter(&connp->conn_lock);
337         error = conn_getsockname(connp, addr, addrlenp);
338         mutex_exit(&connp->conn_lock);

340         return (error);
341 }

343 static int
344 dccp_getsockopt(sock_lower_handle_t proto_handle, int level, int option_name,
345     void *optvalp, socklen_t *optlen, cred_t *cr)
346 {
347         conn_t          *connp = (conn_t *)proto_handle;
348         void            *optvalp_buf;
349         t_uscalar_t     max_optbuf_len;
350         int             len;
351         int             error;

353         cmn_err(CE_NOTE, "dccp_socket.c: dccp_getsockopt");

355         ASSERT(connp->conn_upper_handle != NULL);

357         /* All Solaris components should pass a cred for this operation */
358         ASSERT(cr != NULL);

360         error = proto_opt_check(level, option_name, *optlen, &max_optbuf_len,
361             dccp_opt_obj.odb_opt_des_arr,
362             dccp_opt_obj.odb_opt_arr_cnt,
363             B_FALSE, B_TRUE, cr);
364         if (error != 0) {
365                 if (error < 0) {
366                         error = proto_tlitosyserr(-error);
367                 }
368                 return (error);
369         }

371         optvalp_buf = kmem_alloc(max_optbuf_len, KM_SLEEP);
372         if (optvalp_buf == NULL) {
373                 return (ENOMEM);
374         }

376         error = squeue_synch_enter(connp, NULL);
377         if (error == ENOMEM) {
378                 kmem_free(optvalp_buf, max_optbuf_len);
379                 return (ENOMEM);
380         }

382         len = dccp_opt_get(connp, level, option_name, optvalp_buf);
383         squeue_synch_exit(connp);

385         if (len == -1) {
386                 kmem_free(optvalp_buf, max_optbuf_len);
387                 return (EINVAL);
388         }

390         t_uscalar_t size = MIN(len, *optlen);
```

```
392         bcopy(optvalp_buf, optvalp, size);
393         bcopy(&size, optlen, sizeof (size));

395         kmem_free(optvalp_buf, max_optbuf_len);

397         return (0);
398 }

400 static int
401 dccp_setsockopt(sock_lower_handle_t proto_handle, int level, int option_name,
402     const void *optvalp, socklen_t optlen, cred_t *cr)
403 {
404         conn_t  *connp = (conn_t *)proto_handle;
405         int     error;

407         cmn_err(CE_NOTE, "dccp_socket.c: dccp_setsockopt");

409         ASSERT(connp->conn_upper_handle != NULL);

411         /* All Solaris components should pass a cred for this operation */
412         ASSERT(cr != NULL);

414         error = squeue_synch_enter(connp, NULL);
415         if (error == ENOMEM) {
416                 return (ENOMEM);
417         }

419         error = proto_opt_check(level, option_name, optlen, NULL,
420             dccp_opt_obj.odb_opt_des_arr,
421             dccp_opt_obj.odb_opt_arr_cnt,
422             B_TRUE, B_FALSE, cr);
423         if (error != 0) {
424                 if (error < 0) {
425                         error = proto_tlitosyserr(-error);
426                 }
427                 squeue_synch_exit(connp);
428                 return (error);
429         }

431         error = dccp_opt_set(connp, SETFN_OPTCOM_NEGOTIATE, level, option_name,
432             optlen, (uchar_t *)optvalp, (uint_t *)&optlen, (uchar_t *)optvalp,
433             NULL, cr);
434         squeue_synch_exit(connp);

436         ASSERT(error >= 0);

438         return (error);
439 }

441 /* ARGSUSED */
442 static int
443 dccp_send(sock_lower_handle_t proto_handle, mblk_t *mp, struct nmsghdr *msg,
444     cred_t *cr)
445 {
446         conn_t  *connp = (conn_t *)proto_handle;
447         dccp_t  *dccp;
448         uint32_t msize;
449         int32_t dccpstate;

451         cmn_err(CE_NOTE, "dccp_socket.c: dccp_send");

453         /* All Solaris components should pass a cred for this operation */
454         ASSERT(cr != NULL);

456         ASSERT(connp->conn_ref >= 2);
457         ASSERT(connp->conn_upper_handle != NULL);
```



```
459         if (msg->msg_controllen != 0) {
460                 freemsg(mp);
461                 return (EOPNOTSUPP);
462         }

464         switch (DB_TYPE(mp)) {
465         case M_DATA:
466                 dccp = connp->conn_dccp;
467                 ASSERT(dccp != NULL);

469                 dccpstate = dccp->dccp_state;

471                 /* XXX */

473                 msize = msgdsize(mp);

475                 if (msg->msg_flags & MSG_OOB) {
476                         SQUEUE_ENTER_ONE(connp->conn_sqp, mp, dccp_output_urgent
477                             connp, NULL, dccp_squeue_flag, SQTAG_DCCP_OUTPUT);
478                 } else {
479                         SQUEUE_ENTER_ONE(connp->conn_sqp, mp, dccp_output,
480                             connp, NULL, dccp_squeue_flag, SQTAG_DCCP_OUTPUT);
481                 }

483                 return (0);

485         default:
486                 ASSERT(0);
487         }

489         freemsg(mp);

491         return (0);
492 }

494 /* ARGSUSED */
495 static int
496 dccp_shutdown(sock_lower_handle_t proto_handle, int how, cred_t *cr)
497 {
498         conn_t  *connp = (conn_t *)proto_handle;
499         dccp_t  *dccp = connp->conn_dccp;

501         cmn_err(CE_NOTE, "dccp_socket.c: dccp_shutdown");

503         /* All Solaris components should pass a cred for this operation. */
504         ASSERT(cr != NULL);

506         ASSERT(connp->conn_upper_handle != NULL);


509         return (ENOTSUP);
510 }

512 static void
513 dccp_clr_flowctrl(sock_lower_handle_t proto_handle)
514 {
515         conn_t  *connp = (conn_t *)proto_handle;
516         dccp_t  *dccp = connp->conn_dccp;
517         mblk_t  *mp;
518         int     error;

520         ASSERT(connp->conn_upper_handle != NULL);

522         cmn_err(CE_NOTE, "dccp_socket.c: dccp_clr_flowctrl");
```

```
524         error = squeue_synch_enter(connp, mp);

526         squeue_synch_exit(connp);
527 }

529 /* ARGSUSED */
530 static int
531 dccp_ioctl(sock_lower_handle_t proto_handle, int cmd, intptr_t arg,
532     int mode, int32_t *rvalp, cred_t *cr)
533 {
534         conn_t  *connp = (conn_t *)proto_handle;
535         int     error;

537         cmn_err(CE_NOTE, "dccp_socket.c: dccp_ioctl");

539         ASSERT(connp->conn_upper_handle != NULL);

541         /* All Solaris components should pass a cred for this operation. */
542         ASSERT(cr != NULL);

544         return (ENOTSUP);
545 }

547 /* ARGSUSED */
548 static int
549 dccp_close(sock_lower_handle_t proto_handle, int flags, cred_t *cr)
550 {
551         conn_t  *connp = (conn_t *)proto_handle;

553         cmn_err(CE_NOTE, "dccp_socket.c: dccp_close\n");

555         ASSERT(connp->conn_upper_handle != NULL);

557         /* All Solaris components should pass a cred for this operation */
558         ASSERT(cr != NULL);

560         dccp_close_common(connp, flags);

562         ip_free_helper_stream(connp);

564         CONN_DEC_REF(connp);

566         return (EINPROGRESS);
567 }


570 /*
571  * Socket create function.
572  */
573 sock_lower_handle_t
574 dccp_create(int family, int type, int proto, sock_downcalls_t **sockdowncalls,
575     uint_t *smodep, int *errorp, int flags, cred_t *credp)
576 {
577         conn_t          *connp;
578         boolean_t       isv6;

580         /* XXX (type != SOCK_STREAM */
581         if ((family != AF_INET && family != AF_INET6) ||
582             (proto != 0 && proto != IPPROTO_DCCP)) {
583                 *errorp = EPROTONOSUPPORT;
584                 return (NULL);
585         }

587         cmn_err(CE_NOTE, "dccp_socket: dccp_create\n");

589         isv6 = family == AF_INET6 ? B_TRUE: B_FALSE;
```



```
590         connp = dccp_create_common(credp, isv6, B_TRUE, errorp);
591         if (connp == NULL) {
592                 return (NULL);
593         }

595         /*
596          * Increment ref for DCCP connection.
597          */
598         mutex_enter(&connp->conn_lock);
599         CONN_INC_REF_LOCKED(connp);
600         ASSERT(connp->conn_ref == 2);
601         connp->conn_state_flags &= ~CONN_INCIPIENT;
602         connp->conn_flags |= IPCL_NONSTR;
603         mutex_exit(&connp->conn_lock);

605         ASSERT(errorp != NULL);
606         *errorp = 0;
607         *sockdowncalls = &sock_dccp_downcalls;
608         *smodep = SM_CONNREQUIRED | SM_EXDATA | SM_ACCEPTSUPP |
609             SM_SENDFILESUPP;

611         return ((sock_lower_handle_t)connp);
612 }

614 int
615 dccp_fallback(sock_lower_handle_t proto_handle, queue_t *q,
616     boolean_t issocket, so_proto_quiesced_cb_t quiesced_cb,
617     sock_quiesce_arg_t *arg)
618 {
619         cmn_err(CE_NOTE, "dccp_socket: dccp_fallback\n");

621         return (0);
622 }
623 #endif /* ! codereview */
```

```
**********************************************************
    1436 Mon Jul  9 14:38:13 2012
new/usr/src/uts/common/inet/dccp/dccp_stack.h
dccp: starting module template
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

#ifndef _INET_DCCP_DCCP_STACK_H
#define _INET_DCCP_DCCP_STACK_H

#include <sys/netstack.h>
#include <sys/cpuvar.h>

#include <inet/dccp/dccp_stats.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * DCCP stack instances
 */
typedef struct dccp_stack {
        netstack_t              *dccps_netstack;        /* Common netstack */

        uint_t                  dccps_bind_fanout_size;
        struct dccp_df_s        *dccps_bind_fanout;

        /* Ports */
#define DCCP_NUM_EPRIV_PORTS    64
        int                     dccps_num_epriv_ports;
        in_port_t               dccps_epriv_ports[DCCP_NUM_EPRIV_PORTS];
        kmutex_t                dccps_epriv_port_lock;

        uint_t                  dccps_next_port_to_try;

        in_port_t               dccps_min_anonpriv_port;

        /* Reset rate control */
        int64_t                 dccps_last_rst_intrvl;
        uint32_t                dccps_rst_cnt;

        /* Tunables table */
        struct mod_prop_info_s  *dccps_propinfo_tbl;

        ldi_ident_t             dccps_ldi_ident;

        /* Cpu stats counter */
        dccp_stats_cpu_t        **dccps_sc;
        int                     dccps_sc_cnt;
} dccp_stack_t;

#ifdef  __cplusplus
```



```
}
#endif

#endif  /* _INET_DCCP_DCCP_STACK_H */
#endif /* ! codereview */
```

```
**********************************************************
    6209 Mon Jul  9 14:38:14 2012
new/usr/src/uts/common/inet/dccp/dccp_stats.c
dccp: MIB-II
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * Functions related to MIB-II and kstat.
 */

#include <sys/types.h>
#include <sys/tihdr.h>
#include <sys/policy.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/kstatcom.h>
#include <inet/snmpcom.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"

static int      dccp_snmp_state(dccp_t *);
static int      dccp_kstat_update(kstat_t *, int);

/*
 * Translate DCCP state to MIB2 state.
 */
static int
dccp_snmp_state(dccp_t *dccp)
{
        if (dccp == NULL) {
                return (0);
        }

        switch(dccp->dccp_state) {
        case DCCPS_CLOSED:
                return (MIB2_DCCP_closed);
        default:
```



```
                return (0);
        }
}

/*
 * Get the MIB-II stats.
 */
mblk_t *
dccp_snmp_get(queue_t *q, mblk_t *mpctl, boolean_t legacy_req)
{
        conn_t          *connp = Q_TO_CONN(q);
        connf_t         *connfp;
        ip_stack_t      *ipst;
        dccp_stack_t    *dccps;
        struct opthdr   *optp;
        mblk_t          *mp2ctl;
        mblk_t          *mpdata;
        mblk_t          *mp_conn_ctl = NULL;
        mblk_t          *mp_conn_tail;
        mblk_t          *mp_attr_ctl = NULL;
        mblk_t          *mp_attr_tail;
        mblk_t          *mp6_conn_ctl = NULL;
        mblk_t          *mp6_conn_tail;
        mblk_t          *mp6_attr_ctl = NULL;
        mblk_t          *mp6_attr_tail;
        size_t          dccp_mib_size;
        size_t          dce_size;
        size_t          dce6_size;
        zoneid_t        zoneid;
        int             i;
        mib2_dccp_t             dccp_mib;
        mib2_dccpConnEntry_t    dce;
        mib2_dccp6ConnEntry_t   dce6;

        /*
         * Make a copy of the original message.
         */
        mp2ctl = copymsg(mpctl);

        cmn_err(CE_NOTE, "dccp_stats.c: dccp_snmp_get");

        if (mpctl == NULL ||
            (mpdata = mpctl->b_cont) == NULL ||
            (mp_conn_ctl = copymsg(mpctl)) == NULL ||
            (mp_attr_ctl = copymsg(mpctl)) == NULL) {
                freemsg(mp_conn_ctl);
                freemsg(mp_attr_ctl);
                freemsg(mpctl);
                freemsg(mp2ctl);
                return (NULL);
        }

        ipst = connp->conn_netstack->netstack_ip;
        dccps = connp->conn_netstack->netstack_dccp;

        if (legacy_req) {
                dccp_mib_size = LEGACY_MIB_SIZE(&dccp_mib, mib2_dccp_t);
                dce_size = LEGACY_MIB_SIZE(&dce, mib2_dccpConnEntry_t);
                dce6_size = LEGACY_MIB_SIZE(&dce6, mib2_dccp6ConnEntry_t);
        } else {
                dccp_mib_size = sizeof (mib2_dccp_t);
                dce_size = sizeof (mib2_dccpConnEntry_t);
                dce6_size = sizeof (mib2_dccp6ConnEntry_t);
        }

        bzero(&dccp_mib, sizeof (dccp_mib));
```

```
128         zoneid = Q_TO_CONN(q)->conn_zoneid;
129         mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;

131         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
132                 ipst = dccps->dccps_netstack->netstack_ip;

134                 connfp = &ipst->ips_ipcl_globalhash_fanout[i];
135                 connp = NULL;

137                 while ((connp = ipcl_get_next_conn(connfp, connp,
138                     IPCL_DCCPCONN)) != NULL) {
139                         dccp_t  *dccp;

141                         if (connp->conn_zoneid != zoneid) {
142                                 continue;
143                         }

145                         dccp = connp->conn_dccp;

147                         dce.dccpConnState = dccp_snmp_state(dccp);

149                         if (connp->conn_ipversion == IPV4_VERSION ||
150                             (dccp->dccp_state <= DCCPS_LISTEN)) {

152                                 if (connp->conn_ipversion == IPV6_VERSION) {
153                                         dce.dccpConnRemAddress = INADDR_ANY;
154                                         dce.dccpConnLocalAddress = INADDR_ANY;
155                                 } else {
156                                         dce.dccpConnRemAddress =
157                                             connp->conn_faddr_v4;
158                                         dce.dccpConnLocalAddress =
159                                             connp->conn_laddr_v4;
160                                 }

162                         dce.dccpConnLocalPort = ntohs(connp->conn_lport);
163                         dce.dccpConnRemPort = ntohs(connp->conn_fport);
164 /*
165                         dce.dccpConnCreationProcess = (connp->conn_cpid < 0) ?
166                             MIB2_UNKNOWN_PROCESS : connp->conn_cpid;
167                         dce.dccpConnCreationTime = connp->conn_open_time;
168 */
169                         (void) snmp_append_data2(mp_conn_ctl->b_cont,
170                             &mp_conn_tail, (char *)&dce, dce_size);

172                         }
173                 }
174         }

176         SET_MIB(dccp_mib.dccpConnTableSize, dce_size);
177         SET_MIB(dccp_mib.dccp6ConnTableSize, dce6_size);

179         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
180         optp->level = MIB2_DCCP;
181         optp->name = 0;
182         (void) snmp_append_data(mpdata, (char *)&dccp_mib, dccp_mib_size);
183         optp->len = msgdsize(mpdata);
184         qreply(q, mpctl);

186         optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
187             sizeof (struct T_optmgmt_ack)];
188         optp->level = MIB2_DCCP;
189         optp->name = MIB2_DCCP_CONN;
190         optp->len = msgdsize(mp_conn_ctl->b_cont);
191         qreply(q, mp_conn_ctl);

193         optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
```



```
194             sizeof (struct T_optmgmt_ack)];
195         optp->level = MIB2_DCCP;
196         optp->name = EXPER_XPORT_MLP;
197         optp->len = msgdsize(mp_attr_ctl->b_cont);
198         if (optp->len == 0) {
199                 freemsg(mp_attr_ctl);
200         } else {
201                 qreply(q, mp_attr_ctl);
202         }

204         return (mp2ctl);
205 }

207 /*
208  * DCCP kernel statistics.
209  */
210 void *
211 dccp_kstat_init(netstackid_t stackid)
212 {
213         kstat_t *ksp;

215         dccp_named_kstat_t template = {
216                 { "activeOpens",        KSTAT_DATA_UINT32, 0 },
217                 { "passiveOpens",       KSTAT_DATA_UINT32, 0 },
218                 { "inSegs",             KSTAT_DATA_UINT64, 0 },
219                 { "outSegs",            KSTAT_DATA_UINT64, 0 },
220         };

222         ksp = kstat_create_netstack(DCCP_MOD_NAME, 0, DCCP_MOD_NAME, "mib2",
223             KSTAT_TYPE_NAMED, NUM_OF_FIELDS(dccp_named_kstat_t), 0, stackid);
224         if (ksp == NULL) {
225                 return (NULL);
226         }

228         bcopy(&template, ksp->ks_data, sizeof (template));
229         ksp->ks_update = dccp_kstat_update;
230         ksp->ks_private = (void *)(uintptr_t)stackid;

232         kstat_install(ksp);

234         return (ksp);
235 }

237 /*
238  * Destroy DCCP kernel statistics.
239  */
240 void
241 dccp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
242 {

244         if (ksp != NULL) {
245                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
246                 kstat_delete_netstack(ksp, stackid);
247         }
248 }

250 /*
251  * Update DCCP kernel statistics.
252  */
253 static int
254 dccp_kstat_update(kstat_t *ksp, int rw)
255 {

257         if (rw == KSTAT_WRITE) {
258                 return (EACCES);
259         }
```

```
261         return (0);
262 }
263 #endif /* ! codereview */
```

```
**********************************************************
    1203 Mon Jul  9 14:38:14 2012
new/usr/src/uts/common/inet/dccp/dccp_stats.h
dccp: add dccp_stats.h
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

#ifndef _INET_DCCP_DCCP_STATS_H
#define _INET_DCCP_DCCP_STATS_H

#include <sys/netstack.h>
#include <sys/cpuvar.h>

#ifdef  __cplusplus
extern "C" {
#endif

typedef struct dccp_stat_counter_s {
        uint64_t        dccp_no_listener;
        uint64_t        dccp_listendrop;
        uint64_t        dccp_sock_fallback;
        uint64_t        dccp_rst_unsent;
} dccp_stat_counter_t;

typedef struct {
        uint64_t                dccp_stats_cnt;
        mib2_dccp_t             dccp_sc_mib;
        dccp_stat_counter_t     dccp_sc_stats;
} dccp_stats_cpu_t;

/*
 * MIB-II
 */
#define DCCPS_BUMP_MIB(dccps, x) \
        BUMP_MIB(&(dccps)->dccps_sc[CPU->cpu_seqid]->dccp_sc_mib, x)

#define DCCP_STAT(dccps, x)     \
        ((dccps)->dccps_sc[CPU->cpu_seqid]->dccp_sc_stats.x++)

#ifdef  __cplusplus
}
#endif

#endif  /* _INET_DCCP_DCCP_STATS_H */
#endif /* ! codereview */
```

```
**********************************************************
    5038 Mon Jul  9 14:38:14 2012
new/usr/src/uts/common/inet/dccp/dccp_tpi.c
dccp: clean up
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Functions related to TPI.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/proto_set.h>

#include <sys/cmn_err.h>

#include "dccp_impl.h"

/*
 * This file contains functions related to the TPI interface.
 */

/*
 * XXX
 */
static void
dccp_copy_info(struct T_info_ack *tia, dccp_t *dccp)
{
        conn_t          *connp = dccp->dccp_connp;
        dccp_stack_t    *dccps = dccp->dccp_dccps;
        extern struct T_info_ack dccp_g_t_info_ack;
        extern struct T_info_ack dccp_g_t_info_ack_v6;
```



```
        if (connp->conn_family == AF_INET6) {
                *tia = dccp_g_t_info_ack_v6;
        } else {
                *tia = dccp_g_t_info_ack;
        }

        /* XXX */
}

/*
 * XXX
 */
void
dccp_do_capability_ack(dccp_t *dccp, struct T_capability_ack *tcap,
    t_uscalar_t cap_bits1)
{
        tcap->CAP_bits1 = 0;

        if (cap_bits1 & TC1_INFO) {
                dccp_copy_info(&tcap->INFO_ack, dccp);
                tcap->CAP_bits1 |= TC1_INFO;
        }

        if (cap_bits1 & TC1_ACCEPTOR_ID) {
                tcap->ACCEPTOR_id = dccp->dccp_acceptor_id;
                tcap->CAP_bits1 |= TC1_ACCEPTOR_ID;
        }
}

/*
 * This routine responds to T_CAPABILITY_REQ messages.
 */
void
dccp_capability_req(dccp_t *dccp, mblk_t *mp)
{
        struct T_capability_ack *tcap;
        t_uscalar_t             cap_bits1;

        if (MBLKL(mp) < sizeof (struct T_capability_req)) {
                freemsg(mp);
                return;
        }

        cap_bits1 = ((struct T_capability_req *)mp->b_rptr)->CAP_bits1;

        mp = tpi_ack_alloc(mp, sizeof (struct T_capability_ack),
            mp->b_datap->db_type, T_CAPABILITY_ACK);
        if (mp == NULL) {
                return;
        }

        tcap = (struct T_capability_ack *)mp->b_rptr;
        dccp_do_capability_ack(dccp, tcap, cap_bits1);

        putnext(dccp->dccp_connp->conn_rq, mp);
}

/*
 * Helper function to generate TPI errors acks.
 */
void
dccp_err_ack(dccp_t *dccp, mblk_t *mp, int t_error, int sys_error)
{
        if ((mp = mi_tpi_err_ack_alloc(mp, t_error, sys_error)) != NULL) {
                putnext(dccp->dccp_connp->conn_rq, mp);
```

```
128         }
129 }

131 void
132 dccp_tpi_connect(dccp_t *dccp, mblk_t *mp)
133 {
134         conn_t                  *connp = dccp->dccp_connp;
135         queue_t                 *q = connp->conn_wq;
136         struct sockaddr         *sa;
137         struct T_conn_req       *tcr;
138         sin_t                   *sin;
139         sin6_t                  *sin6;
140         cred_t                  *cr;
141         pid_t                   cpid;
142         socklen_t               len;
143         int                     error;

145         cmn_err(CE_NOTE, "dccp_tpi.c: dccp_tpi_connect");

147         cr = msg_getcred(mp, &cpid);
148         ASSERT(cr != NULL);
149         if (cr == NULL) {
150                 dccp_err_ack(dccp, mp, TSYSERR, EINVAL);
151                 return;
152         }

154         tcr = (struct T_conn_req *)mp->b_rptr;

156         ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <= (uintptr_t)INT_MAX);
157         if ((mp->b_wptr - mp->b_rptr) < sizeof (*tcr)) {
158                 dccp_err_ack(dccp, mp, TPROTO, 0);
159                 return;
160         }

162         error = proto_verify_ip_addr(connp->conn_family, sa, len);
163         if (error != 0) {
164                 dccp_err_ack(dccp, mp, TSYSERR, 0);
165                 return;
166         }

168         error = dccp_do_connect(dccp->dccp_connp, sa, len, cr, cpid);
169         if (error < 0) {
170                 mp = mi_tpi_err_ack_alloc(mp, -error, 0);
171         } else if (error > 0) {
172                 mp = mi_tpi_err_ack_alloc(mp, TSYSERR, error);
173         } else {
174                 mp = mi_tpi_ok_ack_alloc(mp);
175         }
176 }

178 int
179 dccp_tpi_close(queue_t *q, int flags)
180 {
181         conn_t  *connp;

183         ASSERT(WR(q)->q_next == NULL);

185         connp = Q_TO_CONN(q);

187         dccp_close_common(connp, flags);

189         qprocsoff(q);
190         inet_minor_free(connp->conn_minor_arena, connp->conn_dev);

192         return (0);
193 }
```



```
195 int
196 dccp_tpi_close_accept(queue_t *q)
197 {
198         vmem_t  *minor_arena;
199         dev_t   conn_dev;

201         cmn_err(CE_NOTE, "dccp_tpi.c: dccp_tpi_close_accept");

203         return (0);
204 }

206 /*
207  * Options related functions.
208  */
209 int
210 dccp_tpi_opt_get(queue_t *q, int level, int name, uchar_t *ptr)
211 {
212         return (dccp_opt_get(Q_TO_CONN(q), level, name, ptr));
213 }

215 /* ARGSUSED */
216 int
217 dccp_tpi_opt_set(queue_t *q, uint_t optset_context, int level, int name,
218     uint_t inlen, uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
219     void *thisdg_attrs, cred_t *cr)
220 {
221         conn_t  *connp =  Q_TO_CONN(q);

223         return (dccp_opt_set(connp, optset_context, level, name, inlen, invalp,
224             outlenp, outvalp, thisdg_attrs, cr));
225 }

227 void
228 dccp_tpi_accept(queue_t *q, mblk_t *mp)
229 {
230         queue_t         *rq = RD(q);

232         cmn_err(CE_NOTE, "dccp_tpi.c: dccp_tpi_accept");
233 }
234 #endif /* ! codereview */
```

```
**********************************************************
    1647 Mon Jul  9 14:38:14 2012
new/usr/src/uts/common/inet/dccp/dccp_tunables.c
dccp: starting module template
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

/*
 * This file contains tunable properties for DCCP.
 */
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/dccp/dccp_impl.h>
#include <sys/sunddi.h>

mod_prop_info_t dccp_propinfo_tbl[] = {
        /* tunable - 0 */
        { "smallest_nonpriv_port", MOD_PROTO_DCCP,
            mod_set_uint32, mod_get_uint32,
            {1024, (32 * 1024), 1024}, {1024} },

        { "smallest_anon_port", MOD_PROTO_DCCP,
            mod_set_uint32, mod_get_uint32,
            {1024, ULP_MAX_PORT, 32*1024}, {32*1024} },

        { "largest_anon_port", MOD_PROTO_DCCP,
            mod_set_uint32, mod_get_uint32,
            {1024, ULP_MAX_PORT, ULP_MAX_PORT}, {ULP_MAX_PORT} },

        { "_xmit_lowat", MOD_PROTO_DCCP,
            mod_set_uint32, mod_get_uint32,
            {0, (1<<30), DCCP_XMIT_LOWATER},
            {DCCP_XMIT_LOWATER} },

        { "_debug", MOD_PROTO_DCCP,
            mod_set_uint32, mod_get_uint32,
            {0, 10, 0}, {0} },

        { "_rst_sent_rate_enabled", MOD_PROTO_DCCP,
            mod_set_boolean, mod_get_boolean,
            {B_TRUE}, {B_TRUE} },

        { "_rst_sent_rate", MOD_PROTO_DCCP,
            mod_set_uint32, mod_get_uint32,
            {0, UINT32_MAX, 40}, {40} },

        /* tunable - 10 */

        { NULL, 0, NULL, NULL, {0}, {0} }
};

int dccp_propinfo_count = A_CNT(dccp_propinfo_tbl);
#endif /* ! codereview */
```

```
**********************************************************
    1439 Mon Jul  9 14:38:14 2012
new/usr/src/uts/common/inet/dccp/dccpddi.c
dccp: starting module template
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/modctl.h>
#include <inet/common.h>
#include <inet/ip.h>
#include <sys/strsubr.h>
#include <sys/socketvar.h>

#include "dccp_impl.h"

#define INET_NAME       "dccp"
#define INET_MODDESC    "DCCP dummy STREAMS module"
#define INET_DEVDESC    "DCCP STREAMS driver"
#define INET_SOCKDESC   "DCCP socket module"
#define INET_MODSTRTAB  dummymodinfo
#define INET_DEVSTRTAB  dccpinfov4
#define INET_MODMTFLAGS D_MP
#define INET_SOCK_PROTO_CREATE_FUNC     (*dccp_create)
#define INET_SOCK_PROTO_FB_FUNC         (*dccp_fallback)
#define INET_SOCK_FALLBACK_DEV_V4       "/dev/dccp"
#define INET_SOCK_FALLBACK_DEV_V6       "/dev/dccp6"
#define INET_DEVMINOR   0
#define INET_MODMTFLAGS D_MP
#define INET_DEVMTFLAGS (D_MP|_D_DIRECT)

#include "../inetddi.c"

int
_init(void)
{
        return (mod_install(&modlinkage));
}

int
_fini(void)
{
        return (mod_remove(&modlinkage));
}

int
_info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}
#endif /* ! codereview */
```

```
**********************************************************
  140183 Mon Jul  9 14:38:14 2012
new/usr/src/uts/common/inet/ip.h
dccp: starting module template
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright (c) 1990 Mentat Inc.
  25  */

  27 #ifndef _INET_IP_H
  28 #define _INET_IP_H

  30 #ifdef  __cplusplus
  31 extern "C" {
  32 #endif

  34 #include <sys/isa_defs.h>
  35 #include <sys/types.h>
  36 #include <inet/mib2.h>
  37 #include <inet/nd.h>
  38 #include <sys/atomic.h>
  39 #include <net/if_dl.h>
  40 #include <net/if.h>
  41 #include <netinet/ip.h>
  42 #include <netinet/igmp.h>
  43 #include <sys/neti.h>
  44 #include <sys/hook.h>
  45 #include <sys/hook_event.h>
  46 #include <sys/hook_impl.h>
  47 #include <inet/ip_stack.h>

  49 #ifdef _KERNEL
  50 #include <netinet/ip6.h>
  51 #include <sys/avl.h>
  52 #include <sys/list.h>
  53 #include <sys/vmem.h>
  54 #include <sys/squeue.h>
  55 #include <net/route.h>
  56 #include <sys/systm.h>
  57 #include <net/radix.h>
  58 #include <sys/modhash.h>

  60 #ifdef DEBUG
  61 #define CONN_DEBUG
```



```
  62 #endif

  64 #define IP_DEBUG
  65 /*
  66  * The mt-streams(9F) flags for the IP module; put here so that other
  67  * "drivers" that are actually IP (e.g., ICMP, UDP) can use the same set
  68  * of flags.
  69  */
  70 #define IP_DEVMTFLAGS D_MP
  71 #endif  /* _KERNEL */

  73 #define IP_MOD_NAME     "ip"
  74 #define IP_DEV_NAME     "/dev/ip"
  75 #define IP6_DEV_NAME    "/dev/ip6"

  77 #define UDP_MOD_NAME    "udp"
  78 #define UDP_DEV_NAME    "/dev/udp"
  79 #define UDP6_DEV_NAME   "/dev/udp6"

  81 #define TCP_MOD_NAME    "tcp"
  82 #define TCP_DEV_NAME    "/dev/tcp"
  83 #define TCP6_DEV_NAME   "/dev/tcp6"

  85 #define SCTP_MOD_NAME   "sctp"

  87 #define DCCP_MOD_NAME   "dccp"
  88 #define DCCP_DEV_NAME   "/dev/dccp"
  89 #define DCCP6_DEV_NAME  "/dev/dccp6"

  91 #endif /* ! codereview */
  92 #ifndef _IPADDR_T
  93 #define _IPADDR_T
  94 typedef uint32_t ipaddr_t;
  95 #endif

  97 /* Number of bits in an address */
  98 #define IP_ABITS                32
  99 #define IPV4_ABITS              IP_ABITS
 100 #define IPV6_ABITS              128
 101 #define IP_MAX_HW_LEN   40

 103 #define IP_HOST_MASK            (ipaddr_t)0xffffffffU

 105 #define IP_CSUM(mp, off, sum)           (~ip_cksum(mp, off, sum) & 0xFFFF)
 106 #define IP_CSUM_PARTIAL(mp, off, sum)   ip_cksum(mp, off, sum)
 107 #define IP_BCSUM_PARTIAL(bp, len, sum)  bcksum(bp, len, sum)

 109 #define ILL_FRAG_HASH_TBL_COUNT ((unsigned int)64)
 110 #define ILL_FRAG_HASH_TBL_SIZE  (ILL_FRAG_HASH_TBL_COUNT * sizeof (ipfb_t))

 112 #define IPV4_ADDR_LEN                   4
 113 #define IP_ADDR_LEN                     IPV4_ADDR_LEN
 114 #define IP_ARP_PROTO_TYPE               0x0800

 116 #define IPV4_VERSION                    4
 117 #define IP_VERSION                      IPV4_VERSION
 118 #define IP_SIMPLE_HDR_LENGTH_IN_WORDS   5
 119 #define IP_SIMPLE_HDR_LENGTH            20
 120 #define IP_MAX_HDR_LENGTH               60

 122 #define IP_MAX_OPT_LENGTH (IP_MAX_HDR_LENGTH-IP_SIMPLE_HDR_LENGTH)

 124 #define IP_MIN_MTU                      (IP_MAX_HDR_LENGTH + 8) /* 68 bytes */

 126 /*
 127  * XXX IP_MAXPACKET is defined in <netinet/ip.h> as well. At some point the
```

```
 * 2 files should be cleaned up to remove all redundant definitions.
 */
#define IP_MAXPACKET                    65535
#define IP_SIMPLE_HDR_VERSION \
        ((IP_VERSION << 4) | IP_SIMPLE_HDR_LENGTH_IN_WORDS)

#define UDPH_SIZE                       8

/*
 * Constants and type definitions to support IP IOCTL commands
 */
#define IP_IOCTL                        (('i'<<8)|'p')
#define IP_IOC_IRE_DELETE               4
#define IP_IOC_IRE_DELETE_NO_REPLY      5
#define IP_IOC_RTS_REQUEST              7

/* Common definitions used by IP IOCTL data structures */
typedef struct ipllcmd_s {
        uint_t  ipllc_cmd;
        uint_t  ipllc_name_offset;
        uint_t  ipllc_name_length;
} ipllc_t;

/* IP IRE Delete Command Structure. */
typedef struct ipid_s {
        ipllc_t ipid_ipllc;
        uint_t  ipid_ire_type;
        uint_t  ipid_addr_offset;
        uint_t  ipid_addr_length;
        uint_t  ipid_mask_offset;
        uint_t  ipid_mask_length;
} ipid_t;

#define ipid_cmd                ipid_ipllc.ipllc_cmd

#ifdef _KERNEL
/*
 * Temporary state for ip options parser.
 */
typedef struct ipoptp_s
{
        uint8_t         *ipoptp_next;   /* next option to look at */
        uint8_t         *ipoptp_end;    /* end of options */
        uint8_t         *ipoptp_cur;    /* start of current option */
        uint8_t         ipoptp_len;     /* length of current option */
        uint32_t        ipoptp_flags;
} ipoptp_t;

/*
 * Flag(s) for ipoptp_flags
 */
#define IPOPTP_ERROR    0x00000001
#endif  /* _KERNEL */

/* Controls forwarding of IP packets, set via ipadm(1M)/ndd(1M) */
#define IP_FORWARD_NEVER        0
#define IP_FORWARD_ALWAYS       1

#define WE_ARE_FORWARDING(ipst) ((ipst)->ips_ip_forwarding == IP_FORWARD_ALWAYS)

#define IPH_HDR_LENGTH(ipha)                                            \
        ((int)(((ipha_t *)ipha)->ipha_version_and_hdr_length & 0xF) << 2)

#define IPH_HDR_VERSION(ipha)                                           \
        ((int)(((ipha_t *)ipha)->ipha_version_and_hdr_length) >> 4)
```



```
#ifdef _KERNEL
/*
 * IP reassembly macros.  We hide starting and ending offsets in b_next and
 * b_prev of messages on the reassembly queue.  The messages are chained using
 * b_cont.  These macros are used in ip_reassemble() so we don't have to see
 * the ugly casts and assignments.
 * Note that the offsets are <= 64k i.e. a uint_t is sufficient to represent
 * them.
 */
#define IP_REASS_START(mp)              ((uint_t)(uintptr_t)((mp)->b_next))
#define IP_REASS_SET_START(mp, u)       \
        ((mp)->b_next = (mblk_t *)(uintptr_t)(u))
#define IP_REASS_END(mp)                ((uint_t)(uintptr_t)((mp)->b_prev))
#define IP_REASS_SET_END(mp, u)         \
        ((mp)->b_prev = (mblk_t *)(uintptr_t)(u))

#define IP_REASS_COMPLETE       0x1
#define IP_REASS_PARTIAL        0x2
#define IP_REASS_FAILED         0x4

/*
 * Test to determine whether this is a module instance of IP or a
 * driver instance of IP.
 */
#define CONN_Q(q)       (WR(q)->q_next == NULL)

#define Q_TO_CONN(q)    ((conn_t *)(q)->q_ptr)
#define Q_TO_TCP(q)     (Q_TO_CONN((q))->conn_tcp)
#define Q_TO_UDP(q)     (Q_TO_CONN((q))->conn_udp)
#define Q_TO_ICMP(q)    (Q_TO_CONN((q))->conn_icmp)
#define Q_TO_RTS(q)     (Q_TO_CONN((q))->conn_rts)
#define Q_TO_DCCP(q)    (Q_TO_CONN((q))->conn_dccp)
#endif /* ! codereview */

#define CONNP_TO_WQ(connp)      ((connp)->conn_wq)
#define CONNP_TO_RQ(connp)      ((connp)->conn_rq)

#define GRAB_CONN_LOCK(q)       {                               \
        if (q != NULL && CONN_Q(q))                             \
                mutex_enter(&(Q_TO_CONN(q))->conn_lock);        \
}

#define RELEASE_CONN_LOCK(q)    {                               \
        if (q != NULL && CONN_Q(q))                             \
                mutex_exit(&(Q_TO_CONN(q))->conn_lock);         \
}

/*
 * Ref counter macros for ioctls. This provides a guard for TCP to stop
 * tcp_close from removing the rq/wq whilst an ioctl is still in flight on the
 * stream. The ioctl could have been queued on e.g. an ipsq. tcp_close will wait
 * until the ioctlref count is zero before proceeding.
 * Ideally conn_oper_pending_ill would be used for this purpose. However, in the
 * case where an ioctl is aborted or interrupted, it can be cleared prematurely.
 * There are also some race possibilities between ip and the stream head which
 * can also end up with conn_oper_pending_ill being cleared prematurely. So, to
 * avoid these situations, we use a dedicated ref counter for ioctls which is
 * used in addition to and in parallel with the normal conn_ref count.
 */
#define CONN_INC_IOCTLREF_LOCKED(connp) {                       \
        ASSERT(MUTEX_HELD(&(connp)->conn_lock));                \
        DTRACE_PROBE1(conn__inc__ioctlref, conn_t *, (connp));  \
        (connp)->conn_ioctlref++;                               \
        mutex_exit(&(connp)->conn_lock);                        \
}
```

```
#define CONN_INC_IOCTLREF(connp)        {                       \
        mutex_enter(&(connp)->conn_lock);                       \
        CONN_INC_IOCTLREF_LOCKED(connp);                        \
}

#define CONN_DEC_IOCTLREF(connp)        {                       \
        mutex_enter(&(connp)->conn_lock);                       \
        DTRACE_PROBE1(conn__dec__ioctlref, conn_t *, (connp));  \
        /* Make sure conn_ioctlref will not underflow. */       \
        ASSERT((connp)->conn_ioctlref != 0);                    \
        if ((--(connp)->conn_ioctlref == 0) &&                  \
            ((connp)->conn_state_flags & CONN_CLOSING)) {       \
                cv_broadcast(&(connp)->conn_cv);                \
        }                                                       \
        mutex_exit(&(connp)->conn_lock);                        \
}


/*
 * Complete the pending operation. Usually an ioctl. Can also
 * be a bind or option management request that got enqueued
 * in an ipsq_t. Called on completion of the operation.
 */
#define CONN_OPER_PENDING_DONE(connp)   {                       \
        mutex_enter(&(connp)->conn_lock);                       \
        (connp)->conn_oper_pending_ill = NULL;                  \
        cv_broadcast(&(connp)->conn_refcv);                     \
        mutex_exit(&(connp)->conn_lock);                        \
        CONN_DEC_REF(connp);                                    \
}

/*
 * Values for squeue switch:
 */
#define IP_SQUEUE_ENTER_NODRAIN 1
#define IP_SQUEUE_ENTER 2
#define IP_SQUEUE_FILL 3

extern int ip_squeue_flag;

/* IP Fragmentation Reassembly Header */
typedef struct ipf_s {
        struct ipf_s    *ipf_hash_next;
        struct ipf_s    **ipf_ptphn;    /* Pointer to previous hash next. */
        uint32_t        ipf_ident;      /* Ident to match. */
        uint8_t         ipf_protocol;   /* Protocol to match. */
        uchar_t         ipf_last_frag_seen : 1; /* Last fragment seen ? */
        time_t          ipf_timestamp;  /* Reassembly start time. */
        mblk_t          *ipf_mp;        /* mblk we live in. */
        mblk_t          *ipf_tail_mp;   /* Frag queue tail pointer. */
        int             ipf_hole_cnt;   /* Number of holes (hard-case). */
        int             ipf_end;        /* Tail end offset (0 -> hard-case). */
        uint_t          ipf_gen;        /* Frag queue generation */
        size_t          ipf_count;      /* Count of bytes used by frag */
        uint_t          ipf_nf_hdr_len; /* Length of nonfragmented header */
        in6_addr_t      ipf_v6src;      /* IPv6 source address */
        in6_addr_t      ipf_v6dst;      /* IPv6 dest address */
        uint_t          ipf_prev_nexthdr_offset; /* Offset for nexthdr value */
        uint8_t         ipf_ecn;        /* ECN info for the fragments */
        uint8_t         ipf_num_dups;   /* Number of times dup frags recvd */
        uint16_t        ipf_checksum_flags; /* Hardware checksum flags */
        uint32_t        ipf_checksum;   /* Partial checksum of fragment data */
} ipf_t;

/*
 * IPv4 Fragments
```



```
 */
#define IS_V4_FRAGMENT(ipha_fragment_offset_and_flags)                  \
        (((ntohs(ipha_fragment_offset_and_flags) & IPH_OFFSET) != 0) || \
        ((ntohs(ipha_fragment_offset_and_flags) & IPH_MF) != 0))

#define ipf_src V4_PART_OF_V6(ipf_v6src)
#define ipf_dst V4_PART_OF_V6(ipf_v6dst)

#endif /* _KERNEL */

/* ICMP types */
#define ICMP_ECHO_REPLY                 0
#define ICMP_DEST_UNREACHABLE           3
#define ICMP_SOURCE_QUENCH              4
#define ICMP_REDIRECT                   5
#define ICMP_ECHO_REQUEST               8
#define ICMP_ROUTER_ADVERTISEMENT       9
#define ICMP_ROUTER_SOLICITATION        10
#define ICMP_TIME_EXCEEDED              11
#define ICMP_PARAM_PROBLEM              12
#define ICMP_TIME_STAMP_REQUEST         13
#define ICMP_TIME_STAMP_REPLY           14
#define ICMP_INFO_REQUEST               15
#define ICMP_INFO_REPLY                 16
#define ICMP_ADDRESS_MASK_REQUEST       17
#define ICMP_ADDRESS_MASK_REPLY         18

/* Evaluates to true if the ICMP type is an ICMP error */
#define ICMP_IS_ERROR(type)     (               \
        (type) == ICMP_DEST_UNREACHABLE ||      \
        (type) == ICMP_SOURCE_QUENCH ||         \
        (type) == ICMP_TIME_EXCEEDED ||         \
        (type) == ICMP_PARAM_PROBLEM)

/* ICMP_TIME_EXCEEDED codes */
#define ICMP_TTL_EXCEEDED               0
#define ICMP_REASSEMBLY_TIME_EXCEEDED   1

/* ICMP_DEST_UNREACHABLE codes */
#define ICMP_NET_UNREACHABLE            0
#define ICMP_HOST_UNREACHABLE           1
#define ICMP_PROTOCOL_UNREACHABLE       2
#define ICMP_PORT_UNREACHABLE           3
#define ICMP_FRAGMENTATION_NEEDED       4
#define ICMP_SOURCE_ROUTE_FAILED        5
#define ICMP_DEST_NET_UNKNOWN           6
#define ICMP_DEST_HOST_UNKNOWN          7
#define ICMP_SRC_HOST_ISOLATED          8
#define ICMP_DEST_NET_UNREACH_ADMIN     9
#define ICMP_DEST_HOST_UNREACH_ADMIN    10
#define ICMP_DEST_NET_UNREACH_TOS       11
#define ICMP_DEST_HOST_UNREACH_TOS      12

/* ICMP Header Structure */
typedef struct icmph_s {
        uint8_t         icmph_type;
        uint8_t         icmph_code;
        uint16_t        icmph_checksum;
        union {
                struct { /* ECHO request/response structure */
                        uint16_t        u_echo_ident;
                        uint16_t        u_echo_seqnum;
                } u_echo;
                struct { /* Destination unreachable structure */
                        uint16_t        u_du_zero;
                        uint16_t        u_du_mtu;
```

```
392                 } u_du;
393                 struct { /* Parameter problem structure */
394                         uint8_t         u_pp_ptr;
395                         uint8_t         u_pp_rsvd[3];
396                 } u_pp;
397                 struct { /* Redirect structure */
398                         ipaddr_t        u_rd_gateway;
399                 } u_rd;
400         } icmph_u;
401 } icmph_t;

403 #define icmph_echo_ident        icmph_u.u_echo.u_echo_ident
404 #define icmph_echo_seqnum       icmph_u.u_echo.u_echo_seqnum
405 #define icmph_du_zero           icmph_u.u_du.u_du_zero
406 #define icmph_du_mtu            icmph_u.u_du.u_du_mtu
407 #define icmph_pp_ptr            icmph_u.u_pp.u_pp_ptr
408 #define icmph_rd_gateway        icmph_u.u_rd.u_rd_gateway

410 #define ICMPH_SIZE      8

412 /*
413  * Minimum length of transport layer header included in an ICMP error
414  * message for it to be considered valid.
415  */
416 #define ICMP_MIN_TP_HDR_LEN     8

418 /* Aligned IP header */
419 typedef struct ipha_s {
420         uint8_t         ipha_version_and_hdr_length;
421         uint8_t         ipha_type_of_service;
422         uint16_t        ipha_length;
423         uint16_t        ipha_ident;
424         uint16_t        ipha_fragment_offset_and_flags;
425         uint8_t         ipha_ttl;
426         uint8_t         ipha_protocol;
427         uint16_t        ipha_hdr_checksum;
428         ipaddr_t        ipha_src;
429         ipaddr_t        ipha_dst;
430 } ipha_t;

432 /*
433  * IP Flags
434  *
435  * Some of these constant names are copied for the DTrace IP provider in
436  * usr/src/lib/libdtrace/common/{ip.d.in, ip.sed.in}, which should be kept
437  * in sync.
438  */
439 #define IPH_DF          0x4000  /* Don't fragment */
440 #define IPH_MF          0x2000  /* More fragments to come */
441 #define IPH_OFFSET      0x1FFF  /* Where the offset lives */

443 /* Byte-order specific values */
444 #ifdef  _BIG_ENDIAN
445 #define IPH_DF_HTONS    0x4000  /* Don't fragment */
446 #define IPH_MF_HTONS    0x2000  /* More fragments to come */
447 #define IPH_OFFSET_HTONS 0x1FFF /* Where the offset lives */
448 #else
449 #define IPH_DF_HTONS    0x0040  /* Don't fragment */
450 #define IPH_MF_HTONS    0x0020  /* More fragments to come */
451 #define IPH_OFFSET_HTONS 0xFF1F /* Where the offset lives */
452 #endif

454 /* ECN code points for IPv4 TOS byte and IPv6 traffic class octet. */
455 #define IPH_ECN_NECT    0x0     /* Not ECN-Capable Transport */
456 #define IPH_ECN_ECT1    0x1     /* ECN-Capable Transport, ECT(1) */
457 #define IPH_ECN_ECT0    0x2     /* ECN-Capable Transport, ECT(0) */
```



```
458 #define IPH_ECN_CE      0x3     /* ECN-Congestion Experienced (CE) */

460 struct ill_s;

462 typedef void ip_v6intfid_func_t(struct ill_s *, in6_addr_t *);
463 typedef void ip_v6mapinfo_func_t(struct ill_s *, uchar_t *, uchar_t *);
464 typedef void ip_v4mapinfo_func_t(struct ill_s *, uchar_t *, uchar_t *);

466 /* IP Mac info structure */
467 typedef struct ip_m_s {
468         t_uscalar_t             ip_m_mac_type;  /* From <sys/dlpi.h> */
469         int                     ip_m_type;      /* From <net/if_types.h> */
470         t_uscalar_t             ip_m_ipv4sap;
471         t_uscalar_t             ip_m_ipv6sap;
472         ip_v4mapinfo_func_t     *ip_m_v4mapping;
473         ip_v6mapinfo_func_t     *ip_m_v6mapping;
474         ip_v6intfid_func_t      *ip_m_v6intfid;
475         ip_v6intfid_func_t      *ip_m_v6destintfid;
476 } ip_m_t;

478 /*
479  * The following functions attempt to reduce the link layer dependency
480  * of the IP stack. The current set of link specific operations are:
481  * a. map from IPv4 class D (224.0/4) multicast address range or the
482  * IPv6 multicast address range (ff00::/8) to the link layer multicast
483  * address.
484  * b. derive the default IPv6 interface identifier from the interface.
485  * c. derive the default IPv6 destination interface identifier from
486  * the interface (point-to-point only).
487  */
488 extern  void ip_mcast_mapping(struct ill_s *, uchar_t *, uchar_t *);
489 /* ip_m_v6*intfid return void and are never NULL */
490 #define MEDIA_V6INTFID(ip_m, ill, v6ptr) (ip_m)->ip_m_v6intfid(ill, v6ptr)
491 #define MEDIA_V6DESTINTFID(ip_m, ill, v6ptr) \
492         (ip_m)->ip_m_v6destintfid(ill, v6ptr)

494 /* Router entry types */
495 #define IRE_BROADCAST           0x0001  /* Route entry for broadcast address */
496 #define IRE_DEFAULT             0x0002  /* Route entry for default gateway */
497 #define IRE_LOCAL               0x0004  /* Route entry for local address */
498 #define IRE_LOOPBACK            0x0008  /* Route entry for loopback address */
499 #define IRE_PREFIX              0x0010  /* Route entry for prefix routes */
500 #ifndef _KERNEL
501 /* Keep so user-level still compiles */
502 #define IRE_CACHE               0x0020  /* Cached Route entry */
503 #endif
504 #define IRE_IF_NORESOLVER       0x0040  /* Route entry for local interface */
505                                         /* net without any address mapping. */
506 #define IRE_IF_RESOLVER         0x0080  /* Route entry for local interface */
507                                         /* net with resolver. */
508 #define IRE_HOST                0x0100  /* Host route entry */
509 /* Keep so user-level still compiles */
510 #define IRE_HOST_REDIRECT       0x0200  /* only used for T_SVR4_OPTMGMT_REQ */
511 #define IRE_IF_CLONE            0x0400  /* Per host clone of IRE_IF */
512 #define IRE_MULTICAST           0x0800  /* Special - not in table */
513 #define IRE_NOROUTE             0x1000  /* Special - not in table */

515 #define IRE_INTERFACE           (IRE_IF_NORESOLVER | IRE_IF_RESOLVER)

517 #define IRE_IF_ALL              (IRE_IF_NORESOLVER | IRE_IF_RESOLVER | \
518                                     IRE_IF_CLONE)
519 #define IRE_OFFSUBNET           (IRE_DEFAULT | IRE_PREFIX | IRE_HOST)
520 #define IRE_OFFLINK             IRE_OFFSUBNET
521 /*
522  * Note that we view IRE_NOROUTE as ONLINK since we can "send" to them without
523  * going through a router; the result of sending will be an error/icmp error.
```

```
 */
#define IRE_ONLINK              (IRE_IF_ALL|IRE_LOCAL|IRE_LOOPBACK| \
                                    IRE_BROADCAST|IRE_MULTICAST|IRE_NOROUTE)

/* Arguments to ire_flush_cache() */
#define IRE_FLUSH_DELETE        0
#define IRE_FLUSH_ADD           1
#define IRE_FLUSH_GWCHANGE      2

/*
 * Flags to ire_route_recursive
 */
#define IRR_NONE                0
#define IRR_ALLOCATE            1       /* OK to allocate IRE_IF_CLONE */
#define IRR_INCOMPLETE          2       /* OK to return incomplete chain */

/*
 * Open/close synchronization flags.
 * These are kept in a separate field in the conn and the synchronization
 * depends on the atomic 32 bit access to that field.
 */
#define CONN_CLOSING            0x01    /* ip_close waiting for ip_wsrv */
#define CONN_CONDEMNED          0x02    /* conn is closing, no more refs */
#define CONN_INCIPIENT          0x04    /* conn not yet visible, no refs */
#define CONN_QUIESCED           0x08    /* conn is now quiescent */
#define CONN_UPDATE_ILL         0x10    /* conn_update_ill in progress */

/*
 * Flags for dce_flags field. Specifies which information has been set.
 * dce_ident is always present, but the other ones are identified by the flags.
 */
#define DCEF_DEFAULT            0x0001  /* Default DCE - no pmtu or uinfo */
#define DCEF_PMTU               0x0002  /* Different than interface MTU */
#define DCEF_UINFO              0x0004  /* dce_uinfo set */
#define DCEF_TOO_SMALL_PMTU     0x0008  /* Smaller than IPv4/IPv6 MIN */

#ifdef _KERNEL
/*
 * Extra structures need for per-src-addr filtering (IGMPv3/MLDv2)
 */
#define MAX_FILTER_SIZE 64

typedef struct slist_s {
        int             sl_numsrc;
        in6_addr_t      sl_addr[MAX_FILTER_SIZE];
} slist_t;

/*
 * Following struct is used to maintain retransmission state for
 * a multicast group.  One rtx_state_t struct is an in-line field
 * of the ilm_t struct; the slist_ts in the rtx_state_t struct are
 * alloc'd as needed.
 */
typedef struct rtx_state_s {
        uint_t          rtx_timer;      /* retrans timer */
        int             rtx_cnt;        /* retrans count */
        int             rtx_fmode_cnt;  /* retrans count for fmode change */
        slist_t         *rtx_allow;
        slist_t         *rtx_block;
} rtx_state_t;

/*
 * Used to construct list of multicast address records that will be
 * sent in a single listener report.
 */
typedef struct mrec_s {
```



```
        struct mrec_s   *mrec_next;
        uint8_t         mrec_type;
        uint8_t         mrec_auxlen;    /* currently unused */
        in6_addr_t      mrec_group;
        slist_t         mrec_srcs;
} mrec_t;

/* Group membership list per upper conn */

/*
 * We record the multicast information from the socket option in
 * ilg_ifaddr/ilg_ifindex. This allows rejoining the group in the case when
 * the ifaddr (or ifindex) disappears and later reappears, potentially on
 * a different ill. The IPv6 multicast socket options and ioctls all specify
 * the interface using an ifindex. For IPv4 some socket options/ioctls use
 * the interface address and others use the index. We record here the method
 * that was actually used (and leave the other of ilg_ifaddr or ilg_ifindex)
 * at zero so that we can rejoin the way the application intended.
 *
 * We track the ill on which we will or already have joined an ilm using
 * ilg_ill. When we have succeeded joining the ilm and have a refhold on it
 * then we set ilg_ilm. Thus intentionally there is a window where ilg_ill is
 * set and ilg_ilm is not set. This allows clearing ilg_ill as a signal that
 * the ill is being unplumbed and the ilm should be discarded.
 *
 * ilg records the state of multicast memberships of a socket end point.
 * ilm records the state of multicast memberships with the driver and is
 * maintained per interface.
 *
 * The ilg state is protected by conn_ilg_lock.
 * The ilg will not be freed until ilg_refcnt drops to zero.
 */
typedef struct ilg_s {
        struct ilg_s    *ilg_next;
        struct ilg_s    **ilg_ptpn;
        struct conn_s   *ilg_connp;     /* Back pointer to get lock */
        in6_addr_t      ilg_v6group;
        ipaddr_t        ilg_ifaddr;     /* For some IPv4 cases */
        uint_t          ilg_ifindex;    /* IPv6 and some other IPv4 cases */
        struct ill_s    *ilg_ill;       /* Where ilm is joined. No refhold */
        struct ilm_s    *ilg_ilm;       /* With ilm_refhold */
        uint_t          ilg_refcnt;
        mcast_record_t  ilg_fmode;      /* MODE_IS_INCLUDE/MODE_IS_EXCLUDE */
        slist_t         *ilg_filter;
        boolean_t       ilg_condemned;  /* Conceptually deleted */
} ilg_t;

/*
 * Multicast address list entry for ill.
 * ilm_ill is used by IPv4 and IPv6
 *
 * The ilm state (and other multicast state on the ill) is protected by
 * ill_mcast_lock. Operations that change state on both an ilg and ilm
 * in addition use ill_mcast_serializer to ensure that we can't have
 * interleaving between e.g., add and delete operations for the same conn_t,
 * group, and ill. The ill_mcast_serializer is also used to ensure that
 * multicast group joins do not occur on an interface that is in the process
 * of joining an IPMP group.
 *
 * The comment below (and for other netstack_t references) refers
 * to the fact that we only do netstack_hold in particular cases,
 * such as the references from open endpoints (ill_t and conn_t's
 * pointers). Internally within IP we rely on IP's ability to cleanup e.g.
 * ire_t's when an ill goes away.
 */
typedef struct ilm_s {
```

```
656         in6_addr_t      ilm_v6addr;
657         int             ilm_refcnt;
658         uint_t          ilm_timer;      /* IGMP/MLD query resp timer, in msec */
659         struct ilm_s    *ilm_next;      /* Linked list for each ill */
660         uint_t          ilm_state;      /* state of the membership */
661         struct ill_s    *ilm_ill;       /* Back pointer to ill - ill_ilm_cnt */
662         zoneid_t        ilm_zoneid;
663         int             ilm_no_ilg_cnt; /* number of joins w/ no ilg */
664         mcast_record_t  ilm_fmode;      /* MODE_IS_INCLUDE/MODE_IS_EXCLUDE */
665         slist_t         *ilm_filter;    /* source filter list */
666         slist_t         *ilm_pendsrcs;  /* relevant src addrs for pending req */
667         rtx_state_t     ilm_rtx;        /* SCR retransmission state */
668         ipaddr_t        ilm_ifaddr;     /* For IPv4 netstat */
669         ip_stack_t      *ilm_ipst;      /* Does not have a netstack_hold */
670 } ilm_t;

672 #define ilm_addr        V4_PART_OF_V6(ilm_v6addr)

674 /*
675  * Soft reference to an IPsec SA.
676  *
677  * On relative terms, conn's can be persistent (living as long as the
678  * processes which create them), while SA's are ephemeral (dying when
679  * they hit their time-based or byte-based lifetimes).
680  *
681  * We could hold a hard reference to an SA from an ipsec_latch_t,
682  * but this would cause expired SA's to linger for a potentially
683  * unbounded time.
684  *
685  * Instead, we remember the hash bucket number and bucket generation
686  * in addition to the pointer.  The bucket generation is incremented on
687  * each deletion.
688  */
689 typedef struct ipsa_ref_s
690 {
691         struct ipsa_s   *ipsr_sa;
692         struct isaf_s   *ipsr_bucket;
693         uint64_t        ipsr_gen;
694 } ipsa_ref_t;

696 /*
697  * IPsec "latching" state.
698  *
699  * In the presence of IPsec policy, fully-bound conn's bind a connection
700  * to more than just the 5-tuple, but also a specific IPsec action and
701  * identity-pair.
702  * The identity pair is accessed from both the receive and transmit side
703  * hence it is maintained in the ipsec_latch_t structure. conn_latch and
704  * ixa_ipsec_latch points to it.
705  * The policy and actions are stored in conn_latch_in_policy and
706  * conn_latch_in_action for the inbound side, and in ixa_ipsec_policy and
707  * ixa_ipsec_action for the transmit side.
708  *
709  * As an optimization, we also cache soft references to IPsec SA's in
710  * ip_xmit_attr_t so that we can fast-path around most of the work needed for
711  * outbound IPsec SA selection.
712  */
713 typedef struct ipsec_latch_s
714 {
715         kmutex_t        ipl_lock;
716         uint32_t        ipl_refcnt;

718         struct ipsid_s  *ipl_local_cid;
719         struct ipsid_s  *ipl_remote_cid;
720         unsigned int
721                         ipl_ids_latched : 1,
```



```
723                         ipl_pad_to_bit_31 : 31;
724 } ipsec_latch_t;

726 #define IPLATCH_REFHOLD(ipl) { \
727         atomic_add_32(&(ipl)->ipl_refcnt, 1);           \
728         ASSERT((ipl)->ipl_refcnt != 0);                 \
729 }

731 #define IPLATCH_REFRELE(ipl) {                          \
732         ASSERT((ipl)->ipl_refcnt != 0);                         \
733         membar_exit();                                          \
734         if (atomic_add_32_nv(&(ipl)->ipl_refcnt, -1) == 0)      \
735                 iplatch_free(ipl);                              \
736 }

738 /*
739  * peer identity structure.
740  */
741 typedef struct conn_s conn_t;

743 /*
744  * This is used to match an inbound/outbound datagram with policy.
745  */
746 typedef struct ipsec_selector {
747         in6_addr_t      ips_local_addr_v6;
748         in6_addr_t      ips_remote_addr_v6;
749         uint16_t        ips_local_port;
750         uint16_t        ips_remote_port;
751         uint8_t         ips_icmp_type;
752         uint8_t         ips_icmp_code;
753         uint8_t         ips_protocol;
754         uint8_t         ips_isv4 : 1,
755                         ips_is_icmp_inv_acq: 1;
756 } ipsec_selector_t;

758 /*
759  * Note that we put v4 addresses in the *first* 32-bit word of the
760  * selector rather than the last to simplify the prefix match/mask code
761  * in spd.c
762  */
763 #define ips_local_addr_v4 ips_local_addr_v6.s6_addr32[0]
764 #define ips_remote_addr_v4 ips_remote_addr_v6.s6_addr32[0]

766 /* Values used in IP by IPSEC Code */
767 #define         IPSEC_OUTBOUND          B_TRUE
768 #define         IPSEC_INBOUND           B_FALSE

770 /*
771  * There are two variants in policy failures. The packet may come in
772  * secure when not needed (IPSEC_POLICY_???_NOT_NEEDED) or it may not
773  * have the desired level of protection (IPSEC_POLICY_MISMATCH).
774  */
775 #define IPSEC_POLICY_NOT_NEEDED         0
776 #define IPSEC_POLICY_MISMATCH           1
777 #define IPSEC_POLICY_AUTH_NOT_NEEDED    2
778 #define IPSEC_POLICY_ENCR_NOT_NEEDED    3
779 #define IPSEC_POLICY_SE_NOT_NEEDED      4
780 #define IPSEC_POLICY_MAX                5       /* Always max + 1. */

782 /*
783  * Check with IPSEC inbound policy if
784  *
785  * 1) per-socket policy is present - indicated by conn_in_enforce_policy.
786  * 2) Or if we have not cached policy on the conn and the global policy is
787  *    non-empty.
```

```
 */
#define CONN_INBOUND_POLICY_PRESENT(connp, ipss)        \
        ((connp)->conn_in_enforce_policy ||             \
        (!((connp)->conn_policy_cached) &&              \
        (ipss)->ipsec_inbound_v4_policy_present))

#define CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)     \
        ((connp)->conn_in_enforce_policy ||             \
        (!(connp)->conn_policy_cached &&                \
        (ipss)->ipsec_inbound_v6_policy_present))

#define CONN_OUTBOUND_POLICY_PRESENT(connp, ipss)       \
        ((connp)->conn_out_enforce_policy ||            \
        (!((connp)->conn_policy_cached) &&              \
        (ipss)->ipsec_outbound_v4_policy_present))

#define CONN_OUTBOUND_POLICY_PRESENT_V6(connp, ipss)    \
        ((connp)->conn_out_enforce_policy ||            \
        (!(connp)->conn_policy_cached &&                \
        (ipss)->ipsec_outbound_v6_policy_present))

/*
 * Information cached in IRE for upper layer protocol (ULP).
 */
typedef struct iulp_s {
        boolean_t       iulp_set;       /* Is any metric set? */
        uint32_t        iulp_ssthresh;  /* Slow start threshold (TCP). */
        clock_t         iulp_rtt;       /* Guestimate in millisecs. */
        clock_t         iulp_rtt_sd;    /* Cached value of RTT variance. */
        uint32_t        iulp_spipe;     /* Send pipe size. */
        uint32_t        iulp_rpipe;     /* Receive pipe size. */
        uint32_t        iulp_rtomax;    /* Max round trip timeout. */
        uint32_t        iulp_sack;      /* Use SACK option (TCP)? */
        uint32_t        iulp_mtu;       /* Setable with routing sockets */

        uint32_t
                iulp_tstamp_ok : 1,     /* Use timestamp option (TCP)? */
                iulp_wscale_ok : 1,     /* Use window scale option (TCP)? */
                iulp_ecn_ok : 1,        /* Enable ECN (for TCP)? */
                iulp_pmtud_ok : 1,      /* Enable PMTUd? */

                /* These three are passed out by ip_set_destination */
                iulp_localnet: 1,       /* IRE_ONLINK */
                iulp_loopback: 1,       /* IRE_LOOPBACK */
                iulp_local: 1,          /* IRE_LOCAL */

                iulp_not_used : 25;
} iulp_t;

/*
 * The conn drain list structure (idl_t), protected by idl_lock.  Each conn_t
 * inserted in the list points back at this idl_t using conn_idl, and is
 * chained by conn_drain_next and conn_drain_prev, which are also protected by
 * idl_lock.  When flow control is relieved, either ip_wsrv() (STREAMS) or
 * ill_flow_enable() (non-STREAMS) will call conn_drain().
 *
 * The conn drain list, idl_t, itself is part of tx cookie list structure.
 * A tx cookie list points to a blocked Tx ring and contains the list of
 * all conn's that are blocked due to the flow-controlled Tx ring (via
 * the idl drain list). Note that a link can have multiple Tx rings. The
 * drain list will store the conn's blocked due to Tx ring being flow
 * controlled.
 */

typedef uintptr_t ip_mac_tx_cookie_t;
typedef struct idl_s idl_t;
```



```
typedef struct idl_tx_list_s idl_tx_list_t;

struct idl_tx_list_s {
        ip_mac_tx_cookie_t      txl_cookie;
        kmutex_t                txl_lock;       /* Lock for this list */
        idl_t                   *txl_drain_list;
        int                     txl_drain_index;
};

struct idl_s {
        conn_t          *idl_conn;              /* Head of drain list */
        kmutex_t        idl_lock;               /* Lock for this list */
        idl_tx_list_t   *idl_itl;
};

/*
 * Interface route structure which holds the necessary information to recreate
 * routes that are tied to an interface i.e. have ire_ill set.
 *
 * These routes which were initially created via a routing socket or via the
 * SIOCADDRT ioctl may be gateway routes (RTF_GATEWAY being set) or may be
 * traditional interface routes.  When an ill comes back up after being
 * down, this information will be used to recreate the routes.  These
 * are part of an mblk_t chain that hangs off of the ILL (ill_saved_ire_mp).
 */
typedef struct ifrt_s {
        ushort_t        ifrt_type;              /* Type of IRE */
        in6_addr_t      ifrt_v6addr;            /* Address IRE represents. */
        in6_addr_t      ifrt_v6gateway_addr;    /* Gateway if IRE_OFFLINK */
        in6_addr_t      ifrt_v6setsrc_addr;     /* Src addr if RTF_SETSRC */
        in6_addr_t      ifrt_v6mask;            /* Mask for matching IRE. */
        uint32_t        ifrt_flags;             /* flags related to route */
        iulp_t          ifrt_metrics;           /* Routing socket metrics */
        zoneid_t        ifrt_zoneid;            /* zoneid for route */
} ifrt_t;

#define ifrt_addr               V4_PART_OF_V6(ifrt_v6addr)
#define ifrt_gateway_addr       V4_PART_OF_V6(ifrt_v6gateway_addr)
#define ifrt_mask               V4_PART_OF_V6(ifrt_v6mask)
#define ifrt_setsrc_addr        V4_PART_OF_V6(ifrt_v6setsrc_addr)

/* Number of IP addresses that can be hosted on a physical interface */
#define MAX_ADDRS_PER_IF        8192
/*
 * Number of Source addresses to be considered for source address
 * selection. Used by ipif_select_source_v4/v6.
 */
#define MAX_IPIF_SELECT_SOURCE  50

#ifdef IP_DEBUG
/*
 * Trace refholds and refreles for debugging.
 */
#define TR_STACK_DEPTH  14
typedef struct tr_buf_s {
        int     tr_depth;
        clock_t tr_time;
        pc_t    tr_stack[TR_STACK_DEPTH];
} tr_buf_t;

typedef struct th_trace_s {
        int             th_refcnt;
        uint_t          th_trace_lastref;
        kthread_t       *th_id;
#define TR_BUF_MAX      38
        tr_buf_t        th_trbuf[TR_BUF_MAX];
```

```
} th_trace_t;

typedef struct th_hash_s {
        list_node_t     thh_link;
        mod_hash_t      *thh_hash;
        ip_stack_t      *thh_ipst;
} th_hash_t;
#endif

/* The following are ipif_state_flags */
#define IPIF_CONDEMNED          0x1     /* The ipif is being removed */
#define IPIF_CHANGING           0x2     /* A critcal ipif field is changing */
#define IPIF_SET_LINKLOCAL      0x10    /* transient flag during bringup */

/* IP interface structure, one per local address */
typedef struct ipif_s {
        struct  ipif_s  *ipif_next;
        struct  ill_s   *ipif_ill;      /* Back pointer to our ill */
        int     ipif_id;                /* Logical unit number */
        in6_addr_t ipif_v6lcl_addr;     /* Local IP address for this if. */
        in6_addr_t ipif_v6subnet;       /* Subnet prefix for this if. */
        in6_addr_t ipif_v6net_mask;     /* Net mask for this interface. */
        in6_addr_t ipif_v6brd_addr;     /* Broadcast addr for this interface. */
        in6_addr_t ipif_v6pp_dst_addr;  /* Point-to-point dest address. */
        uint64_t ipif_flags;            /* Interface flags. */
        uint_t  ipif_ire_type;          /* IRE_LOCAL or IRE_LOOPBACK */

        /*
         * The packet count in the ipif contain the sum of the
         * packet counts in dead IRE_LOCAL/LOOPBACK for this ipif.
         */
        uint_t  ipif_ib_pkt_count;      /* Inbound packets for our dead IREs */

        /* Exclusive bit fields, protected by ipsq_t */
        unsigned int
                ipif_was_up : 1,        /* ipif was up before */
                ipif_addr_ready : 1,    /* DAD is done */
                ipif_was_dup : 1,       /* DAD had failed */
                ipif_added_nce : 1,     /* nce added for local address */

                ipif_pad_to_31 : 28;

        ilm_t   *ipif_allhosts_ilm;     /* For all-nodes join */
        ilm_t   *ipif_solmulti_ilm;     /* For IPv6 solicited multicast join */

        uint_t  ipif_seqid;             /* unique index across all ills */
        uint_t  ipif_state_flags;       /* See IPIF_* flag defs above */
        uint_t  ipif_refcnt;            /* active consistent reader cnt */

        zoneid_t ipif_zoneid;           /* zone ID number */
        timeout_id_t ipif_recovery_id;  /* Timer for DAD recovery */
        boolean_t ipif_trace_disable;   /* True when alloc fails */
        /*
         * For an IPMP interface, ipif_bound_ill tracks the ill whose hardware
         * information this ipif is associated with via ARP/NDP.  We can use
         * an ill pointer (rather than an index) because only ills that are
         * part of a group will be pointed to, and an ill cannot disappear
         * while it's in a group.
         */
        struct ill_s    *ipif_bound_ill;
        struct ipif_s   *ipif_bound_next; /* bound ipif chain */
        boolean_t       ipif_bound;     /* B_TRUE if we successfully bound */

        struct ire_s    *ipif_ire_local; /* Our IRE_LOCAL or LOOPBACK */
        struct ire_s    *ipif_ire_if;    /* Our IRE_INTERFACE */
} ipif_t;
```



```
/*
 * The following table lists the protection levels of the various members
 * of the ipif_t. The following notation is used.
 *
 * Write once - Written to only once at the time of bringing up
 * the interface and can be safely read after the bringup without any lock.
 *
 * ipsq - Need to execute in the ipsq to perform the indicated access.
 *
 * ill_lock - Need to hold this mutex to perform the indicated access.
 *
 * ill_g_lock - Need to hold this rw lock as reader/writer for read access or
 * write access respectively.
 *
 * down ill - Written to only when the ill is down (i.e all ipifs are down)
 * up ill - Read only when the ill is up (i.e. at least 1 ipif is up)
 *
 *               Table of ipif_t members and their protection
 *
 * ipif_next            ipsq + ill_lock +       ipsq OR ill_lock OR
 *                      ill_g_lock              ill_g_lock
 * ipif_ill             ipsq + down ipif        write once
 * ipif_id              ipsq + down ipif        write once
 * ipif_v6lcl_addr      ipsq + down ipif        up ipif
 * ipif_v6subnet        ipsq + down ipif        up ipif
 * ipif_v6net_mask      ipsq + down ipif        up ipif
 *
 * ipif_v6brd_addr
 * ipif_v6pp_dst_addr
 * ipif_flags           ill_lock                ill_lock
 * ipif_ire_type        ipsq + down ill         up ill
 *
 * ipif_ib_pkt_count    Approx
 *
 * bit fields           ill_lock                ill_lock
 *
 * ipif_allhosts_ilm    ipsq                    ipsq
 * ipif_solmulti_ilm    ipsq                    ipsq
 *
 * ipif_seqid           ipsq                    Write once
 *
 * ipif_state_flags     ill_lock                ill_lock
 * ipif_refcnt          ill_lock                ill_lock
 * ipif_bound_ill       ipsq + ipmp_lock        ipsq OR ipmp_lock
 * ipif_bound_next      ipsq                    ipsq
 * ipif_bound           ipsq                    ipsq
 *
 * ipif_ire_local       ipsq + ips_ill_g_lock   ipsq OR ips_ill_g_lock
 * ipif_ire_if          ipsq + ips_ill_g_lock   ipsq OR ips_ill_g_lock
 */

/*
 * Return values from ip_laddr_verify_{v4,v6}
 */
typedef enum { IPVL_UNICAST_UP, IPVL_UNICAST_DOWN, IPVL_MCAST, IPVL_BCAST,
               IPVL_BAD} ip_laddr_t;


#define IP_TR_HASH(tid) ((((uintptr_t)tid) >> 6) & (IP_TR_HASH_MAX - 1))

#ifdef DEBUG
#define IPIF_TRACE_REF(ipif)    ipif_trace_ref(ipif)
#define ILL_TRACE_REF(ill)      ill_trace_ref(ill)
#define IPIF_UNTRACE_REF(ipif)  ipif_untrace_ref(ipif)
#define ILL_UNTRACE_REF(ill)    ill_untrace_ref(ill)
```

```
#else
#define IPIF_TRACE_REF(ipif)
#define ILL_TRACE_REF(ill)
#define IPIF_UNTRACE_REF(ipif)
#define ILL_UNTRACE_REF(ill)
#endif

/* IPv4 compatibility macros */
#define ipif_lcl_addr           V4_PART_OF_V6(ipif_v6lcl_addr)
#define ipif_subnet             V4_PART_OF_V6(ipif_v6subnet)
#define ipif_net_mask           V4_PART_OF_V6(ipif_v6net_mask)
#define ipif_brd_addr           V4_PART_OF_V6(ipif_v6brd_addr)
#define ipif_pp_dst_addr        V4_PART_OF_V6(ipif_v6pp_dst_addr)

/* Macros for easy backreferences to the ill. */
#define ipif_isv6               ipif_ill->ill_isv6

#define SIOCLIFADDR_NDX 112     /* ndx of SIOCLIFADDR in the ndx ioctl table */

/*
 * mode value for ip_ioctl_finish for finishing an ioctl
 */
#define CONN_CLOSE      1               /* No mi_copy */
#define COPYOUT         2               /* do an mi_copyout if needed */
#define NO_COPYOUT      3               /* do an mi_copy_done */
#define IPI2MODE(ipi)   ((ipi)->ipi_flags & IPI_GET_CMD ? COPYOUT : NO_COPYOUT)

/*
 * The IP-MT design revolves around the serialization objects ipsq_t (IPSQ)
 * and ipxop_t (exclusive operation or "xop").  Becoming "writer" on an IPSQ
 * ensures that no other threads can become "writer" on any IPSQs sharing that
 * IPSQ's xop until the writer thread is done.
 *
 * Each phyint points to one IPSQ that remains fixed over the phyint's life.
 * Each IPSQ points to one xop that can change over the IPSQ's life.  If a
 * phyint is *not* in an IPMP group, then its IPSQ will refer to the IPSQ's
 * "own" xop (ipsq_ownxop).  If a phyint *is* part of an IPMP group, then its
 * IPSQ will refer to the "group" xop, which is shorthand for the xop of the
 * IPSQ of the IPMP meta-interface's phyint.  Thus, all phyints that are part
 * of the same IPMP group will have their IPSQ's point to the group xop, and
 * thus becoming "writer" on any phyint in the group will prevent any other
 * writer on any other phyint in the group.  All IPSQs sharing the same xop
 * are chained together through ipsq_next (in the degenerate common case,
 * ipsq_next simply refers to itself).  Note that the group xop is guaranteed
 * to exist at least as long as there are members in the group, since the IPMP
 * meta-interface can only be destroyed if the group is empty.
 *
 * Incoming exclusive operation requests are enqueued on the IPSQ they arrived
 * on rather than the xop.  This makes switching xop's (as would happen when a
 * phyint leaves an IPMP group) simple, because after the phyint leaves the
 * group, any operations enqueued on its IPSQ can be safely processed with
 * respect to its new xop, and any operations enqueued on the IPSQs of its
 * former group can be processed with respect to their existing group xop.
 * Even so, switching xops is a subtle dance; see ipsq_dq() for details.
 *
 * An IPSQ's "own" xop is embedded within the IPSQ itself since they have have
 * identical lifetimes, and because doing so simplifies pointer management.
 * While each phyint and IPSQ point to each other, it is not possible to free
 * the IPSQ when the phyint is freed, since we may still *inside* the IPSQ
 * when the phyint is being freed.  Thus, ipsq_phyint is set to NULL when the
 * phyint is freed, and the IPSQ free is later done in ipsq_exit().
 *
 * ipsq_t synchronization:      read                    write
 *
 *      ipsq_xopq_mphead        ipx_lock                ipx_lock
 *      ipsq_xopq_mptail        ipx_lock                ipx_lock
```



```
 *      ipsq_xop_switch_mp      ipsq_lock               ipsq_lock
 *      ipsq_phyint             write once              write once
 *      ipsq_next               RW_READER ill_g_lock    RW_WRITER ill_g_lock
 *      ipsq_xop                ipsq_lock or ipsq       ipsq_lock + ipsq
 *      ipsq_swxop              ipsq                    ipsq
 *      ipsq_ownxop             see ipxop_t             see ipxop_t
 *      ipsq_ipst               write once              write once
 *
 * ipxop_t synchronization:     read                    write
 *
 *      ipx_writer              ipx_lock                ipx_lock
 *      ipx_xop_queued          ipx_lock                ipx_lock
 *      ipx_mphead              ipx_lock                ipx_lock
 *      ipx_mptail              ipx_lock                ipx_lock
 *      ipx_ipsq                write once              write once
 *      ips_ipsq_queued         ipx_lock                ipx_lock
 *      ipx_waitfor             ipsq or ipx_lock        ipsq + ipx_lock
 *      ipx_reentry_cnt         ipsq or ipx_lock        ipsq + ipx_lock
 *      ipx_current_done        ipsq                    ipsq
 *      ipx_current_ioctl       ipsq                    ipsq
 *      ipx_current_ipif        ipsq or ipx_lock        ipsq + ipx_lock
 *      ipx_pending_ipif        ipsq or ipx_lock        ipsq + ipx_lock
 *      ipx_pending_mp          ipsq or ipx_lock        ipsq + ipx_lock
 *      ipx_forced              ipsq                    ipsq
 *      ipx_depth               ipsq                    ipsq
 *      ipx_stack               ipsq                    ipsq
 */
typedef struct ipxop_s {
        kmutex_t        ipx_lock;       /* see above */
        kthread_t       *ipx_writer;    /* current owner */
        mblk_t          *ipx_mphead;    /* messages tied to this op */
        mblk_t          *ipx_mptail;
        struct ipsq_s   *ipx_ipsq;      /* associated ipsq */
        boolean_t       ipx_ipsq_queued; /* ipsq using xop has queued op */
        int             ipx_waitfor;    /* waiting; values encoded below */
        int             ipx_reentry_cnt;
        boolean_t       ipx_current_done;  /* is the current operation done? */
        int             ipx_current_ioctl; /* current ioctl, or 0 if no ioctl */
        ipif_t          *ipx_current_ipif; /* ipif for current op */
        ipif_t          *ipx_pending_ipif; /* ipif for ipx_pending_mp */
        mblk_t          *ipx_pending_mp;   /* current ioctl mp while waiting */
        boolean_t       ipx_forced;                     /* debugging aid */
#ifdef DEBUG
        int             ipx_depth;                      /* debugging aid */
#define IPX_STACK_DEPTH 15
        pc_t            ipx_stack[IPX_STACK_DEPTH];     /* debugging aid */
#endif
} ipxop_t;

typedef struct ipsq_s {
        kmutex_t ipsq_lock;             /* see above */
        mblk_t  *ipsq_switch_mp;        /* op to handle right after switch */
        mblk_t  *ipsq_xopq_mphead;      /* list of excl ops (mostly ioctls) */
        mblk_t  *ipsq_xopq_mptail;
        struct phyint   *ipsq_phyint;   /* associated phyint */
        struct ipsq_s   *ipsq_next;     /* next ipsq sharing ipsq_xop */
        struct ipxop_s  *ipsq_xop;      /* current xop synchronization info */
        struct ipxop_s  *ipsq_swxop;    /* switch xop to on ipsq_exit() */
        struct ipxop_s  ipsq_ownxop;    /* our own xop (may not be in-use) */
        ip_stack_t      *ipsq_ipst;     /* does not have a netstack_hold */
} ipsq_t;

/*
 * ipx_waitfor values:
 */
enum {
```

```
1184         IPIF_DOWN = 1,  /* ipif_down() waiting for refcnts to drop */
1185         ILL_DOWN,       /* ill_down() waiting for refcnts to drop */
1186         IPIF_FREE,      /* ipif_free() waiting for refcnts to drop */
1187         ILL_FREE        /* ill unplumb waiting for refcnts to drop */
1188 };

1190 /* Operation types for ipsq_try_enter() */
1191 #define CUR_OP 0        /* request writer within current operation */
1192 #define NEW_OP 1        /* request writer for a new operation */
1193 #define SWITCH_OP 2     /* request writer once IPSQ XOP switches */

1195 /*
1196  * Kstats tracked on each IPMP meta-interface.  Order here must match
1197  * ipmp_kstats[] in ip/ipmp.c.
1198  */
1199 enum {
1200         IPMP_KSTAT_OBYTES,      IPMP_KSTAT_OBYTES64,    IPMP_KSTAT_RBYTES,
1201         IPMP_KSTAT_RBYTES64,    IPMP_KSTAT_OPACKETS,    IPMP_KSTAT_OPACKETS64,
1202         IPMP_KSTAT_OERRORS,     IPMP_KSTAT_IPACKETS,    IPMP_KSTAT_IPACKETS64,
1203         IPMP_KSTAT_IERRORS,     IPMP_KSTAT_MULTIRCV,    IPMP_KSTAT_MULTIXMT,
1204         IPMP_KSTAT_BRDCSTRCV,   IPMP_KSTAT_BRDCSTXMT,   IPMP_KSTAT_LINK_UP,
1205         IPMP_KSTAT_MAX          /* keep last */
1206 };

1208 /*
1209  * phyint represents state that is common to both IPv4 and IPv6 interfaces.
1210  * There is a separate ill_t representing IPv4 and IPv6 which has a
1211  * backpointer to the phyint structure for accessing common state.
1212  */
1213 typedef struct phyint {
1214         struct ill_s    *phyint_illv4;
1215         struct ill_s    *phyint_illv6;
1216         uint_t          phyint_ifindex;         /* SIOCSLIFINDEX */
1217         uint64_t        phyint_flags;
1218         avl_node_t      phyint_avl_by_index;    /* avl tree by index */
1219         avl_node_t      phyint_avl_by_name;     /* avl tree by name */
1220         kmutex_t        phyint_lock;
1221         struct ipsq_s   *phyint_ipsq;           /* back pointer to ipsq */
1222         struct ipmp_grp_s *phyint_grp;          /* associated IPMP group */
1223         char            phyint_name[LIFNAMSIZ]; /* physical interface name */
1224         uint64_t        phyint_kstats0[IPMP_KSTAT_MAX]; /* baseline kstats */
1225 } phyint_t;

1227 #define CACHE_ALIGN_SIZE 64
1228 #define CACHE_ALIGN(align_struct)       P2ROUNDUP(sizeof (struct align_struct),\
1229                                                 CACHE_ALIGN_SIZE)
1230 struct _phyint_list_s_ {
1231         avl_tree_t      phyint_list_avl_by_index;       /* avl tree by index */
1232         avl_tree_t      phyint_list_avl_by_name;        /* avl tree by name */
1233 };

1235 typedef union phyint_list_u {
1236         struct  _phyint_list_s_ phyint_list_s;
1237         char    phyint_list_filler[CACHE_ALIGN(_phyint_list_s_)];
1238 } phyint_list_t;

1240 #define phyint_list_avl_by_index        phyint_list_s.phyint_list_avl_by_index
1241 #define phyint_list_avl_by_name         phyint_list_s.phyint_list_avl_by_name

1243 /*
1244  * Fragmentation hash bucket
1245  */
1246 typedef struct ipfb_s {
1247         struct ipf_s    *ipfb_ipf;      /* List of ... */
1248         size_t          ipfb_count;     /* Count of bytes used by frag(s) */
1249         kmutex_t        ipfb_lock;      /* Protect all ipf in list */
```



```
1250         uint_t          ipfb_frag_pkts; /* num of distinct fragmented pkts */
1251 } ipfb_t;

1253 /*
1254  * IRE bucket structure. Usually there is an array of such structures,
1255  * each pointing to a linked list of ires. irb_refcnt counts the number
1256  * of walkers of a given hash bucket. Usually the reference count is
1257  * bumped up if the walker wants no IRES to be DELETED while walking the
1258  * list. Bumping up does not PREVENT ADDITION. This allows walking a given
1259  * hash bucket without stumbling up on a free pointer.
1260  *
1261  * irb_t structures in ip_ftable are dynamically allocated and freed.
1262  * In order to identify the irb_t structures that can be safely kmem_free'd
1263  * we need to ensure that
1264  *  - the irb_refcnt is quiescent, indicating no other walkers,
1265  *  - no other threads or ire's are holding references to the irb,
1266  *      i.e., irb_nire == 0,
1267  *  - there are no active ire's in the bucket, i.e., irb_ire_cnt == 0
1268  */
1269 typedef struct irb {
1270         struct ire_s    *irb_ire;       /* First ire in this bucket */
1271                                         /* Should be first in this struct */
1272         krwlock_t       irb_lock;       /* Protect this bucket */
1273         uint_t          irb_refcnt;     /* Protected by irb_lock */
1274         uchar_t         irb_marks;      /* CONDEMNED ires in this bucket ? */
1275 #define IRB_MARK_CONDEMNED      0x0001  /* Contains some IRE_IS_CONDEMNED */
1276 #define IRB_MARK_DYNAMIC        0x0002  /* Dynamically allocated */
1277         /* Once IPv6 uses radix then IRB_MARK_DYNAMIC will be always be set */
1278         uint_t          irb_ire_cnt;    /* Num of active IRE in this bucket */
1279         int             irb_nire;       /* Num of ftable ire's that ref irb */
1280         ip_stack_t      *irb_ipst;      /* Does not have a netstack_hold */
1281 } irb_t;

1283 /*
1284  * This is the structure used to store the multicast physical addresses
1285  * that an interface has joined.
1286  * The refcnt keeps track of the number of multicast IP addresses mapping
1287  * to a physical multicast address.
1288  */
1289 typedef struct multiphysaddr_s {
1290         struct  multiphysaddr_s  *mpa_next;
1291         char    mpa_addr[IP_MAX_HW_LEN];
1292         int     mpa_refcnt;
1293 } multiphysaddr_t;

1295 #define IRB2RT(irb)     (rt_t *)((caddr_t)(irb) - offsetof(rt_t, rt_irb))

1297 /* Forward declarations */
1298 struct dce_s;
1299 typedef struct dce_s dce_t;
1300 struct ire_s;
1301 typedef struct ire_s ire_t;
1302 struct ncec_s;
1303 typedef struct ncec_s ncec_t;
1304 struct nce_s;
1305 typedef struct nce_s nce_t;
1306 struct ip_recv_attr_s;
1307 typedef struct ip_recv_attr_s ip_recv_attr_t;
1308 struct ip_xmit_attr_s;
1309 typedef struct ip_xmit_attr_s ip_xmit_attr_t;

1311 struct tsol_ire_gw_secattr_s;
1312 typedef struct tsol_ire_gw_secattr_s tsol_ire_gw_secattr_t;

1314 /*
1315  * This is a structure for a one-element route cache that is passed
```

```
 * by reference between ip_input and ill_inputfn.
 */
typedef struct {
        ire_t           *rtc_ire;
        ipaddr_t        rtc_ipaddr;
        in6_addr_t      rtc_ip6addr;
} rtc_t;

/*
 * Note: Temporarily use 64 bits, and will probably go back to 32 bits after
 * more cleanup work is done.
 */
typedef uint64_t iaflags_t;

/* The ill input function pointer type */
typedef void (*pfillinput_t)(mblk_t *, void *, void *, ip_recv_attr_t *,
    rtc_t *);

/* The ire receive function pointer type */
typedef void (*pfirerecv_t)(ire_t *, mblk_t *, void *, ip_recv_attr_t *);

/* The ire send and postfrag function pointer types */
typedef int (*pfiresend_t)(ire_t *, mblk_t *, void *,
    ip_xmit_attr_t *, uint32_t *);
typedef int (*pfirepostfrag_t)(mblk_t *, nce_t *, iaflags_t, uint_t, uint32_t,
    zoneid_t, zoneid_t, uintptr_t *);


#define IP_V4_G_HEAD    0
#define IP_V6_G_HEAD    1

#define MAX_G_HEADS     2

/*
 * unpadded ill_if structure
 */
struct  _ill_if_s_ {
        union ill_if_u  *illif_next;
        union ill_if_u  *illif_prev;
        avl_tree_t      illif_avl_by_ppa;       /* AVL tree sorted on ppa */
        vmem_t          *illif_ppa_arena;       /* ppa index space */
        uint16_t        illif_mcast_v1;         /* hints for            */
        uint16_t        illif_mcast_v2;         /* [igmp|mld]_slowtimo    */
        int             illif_name_len;         /* name length */
        char            illif_name[LIFNAMSIZ];  /* name of interface type */
};

/* cache aligned ill_if structure */
typedef union   ill_if_u {
        struct  _ill_if_s_ ill_if_s;
        char    illif_filler[CACHE_ALIGN(_ill_if_s_)];
} ill_if_t;

#define illif_next              ill_if_s.illif_next
#define illif_prev              ill_if_s.illif_prev
#define illif_avl_by_ppa        ill_if_s.illif_avl_by_ppa
#define illif_ppa_arena         ill_if_s.illif_ppa_arena
#define illif_mcast_v1          ill_if_s.illif_mcast_v1
#define illif_mcast_v2          ill_if_s.illif_mcast_v2
#define illif_name              ill_if_s.illif_name
#define illif_name_len          ill_if_s.illif_name_len

typedef struct ill_walk_context_s {
        int     ctx_current_list; /* current list being searched */
        int     ctx_last_list;   /* last list to search */
} ill_walk_context_t;
```



```
/*
 * ill_g_heads structure, one for IPV4 and one for IPV6
 */
struct _ill_g_head_s_ {
        ill_if_t        *ill_g_list_head;
        ill_if_t        *ill_g_list_tail;
};

typedef union ill_g_head_u {
        struct _ill_g_head_s_ ill_g_head_s;
        char    ill_g_head_filler[CACHE_ALIGN(_ill_g_head_s_)];
} ill_g_head_t;

#define ill_g_list_head ill_g_head_s.ill_g_list_head
#define ill_g_list_tail ill_g_head_s.ill_g_list_tail

#define IP_V4_ILL_G_LIST(ipst)  \
        (ipst)->ips_ill_g_heads[IP_V4_G_HEAD].ill_g_list_head
#define IP_V6_ILL_G_LIST(ipst)  \
        (ipst)->ips_ill_g_heads[IP_V6_G_HEAD].ill_g_list_head
#define IP_VX_ILL_G_LIST(i, ipst)       \
        (ipst)->ips_ill_g_heads[i].ill_g_list_head

#define ILL_START_WALK_V4(ctx_ptr, ipst)        \
        ill_first(IP_V4_G_HEAD, IP_V4_G_HEAD, ctx_ptr, ipst)
#define ILL_START_WALK_V6(ctx_ptr, ipst)        \
        ill_first(IP_V6_G_HEAD, IP_V6_G_HEAD, ctx_ptr, ipst)
#define ILL_START_WALK_ALL(ctx_ptr, ipst)       \
        ill_first(MAX_G_HEADS, MAX_G_HEADS, ctx_ptr, ipst)

/*
 * Capabilities, possible flags for ill_capabilities.
 */
#define ILL_CAPAB_LSO           0x04            /* Large Send Offload */
#define ILL_CAPAB_HCKSUM        0x08            /* Hardware checksumming */
#define ILL_CAPAB_ZEROCOPY      0x10            /* Zero-copy */
#define ILL_CAPAB_DLD           0x20            /* DLD capabilities */
#define ILL_CAPAB_DLD_POLL      0x40            /* Polling */
#define ILL_CAPAB_DLD_DIRECT    0x80            /* Direct function call */

/*
 * Per-ill Hardware Checksumming capbilities.
 */
typedef struct ill_hcksum_capab_s ill_hcksum_capab_t;

/*
 * Per-ill Zero-copy capabilities.
 */
typedef struct ill_zerocopy_capab_s ill_zerocopy_capab_t;

/*
 * DLD capbilities.
 */
typedef struct ill_dld_capab_s ill_dld_capab_t;

/*
 * Per-ill polling resource map.
 */
typedef struct ill_rx_ring ill_rx_ring_t;

/*
 * Per-ill Large Send Offload capabilities.
 */
typedef struct ill_lso_capab_s ill_lso_capab_t;
```

```
/* The following are ill_state_flags */
#define ILL_LL_SUBNET_PENDING   0x01    /* Waiting for DL_INFO_ACK from drv */
#define ILL_CONDEMNED           0x02    /* No more new ref's to the ILL */
#define ILL_DL_UNBIND_IN_PROGRESS       0x04    /* UNBIND_REQ is sent */
/*
 * ILL_DOWN_IN_PROGRESS is set to ensure the following:
 * - no packets are sent to the driver after the DL_UNBIND_REQ is sent,
 * - no longstanding references will be acquired on objects that are being
 *   brought down.
 */
#define ILL_DOWN_IN_PROGRESS    0x08

/* Is this an ILL whose source address is used by other ILL's ? */
#define IS_USESRC_ILL(ill)                      \
        (((ill)->ill_usesrc_ifindex == 0) &&    \
        ((ill)->ill_usesrc_grp_next != NULL))

/* Is this a client/consumer of the usesrc ILL ? */
#define IS_USESRC_CLI_ILL(ill)                  \
        (((ill)->ill_usesrc_ifindex != 0) &&    \
        ((ill)->ill_usesrc_grp_next != NULL))

/* Is this an virtual network interface (vni) ILL ? */
#define IS_VNI(ill)                                                     \
        (((ill)->ill_phyint->phyint_flags & (PHYI_LOOPBACK|PHYI_VIRTUAL)) == \
        PHYI_VIRTUAL)

/* Is this a loopback ILL? */
#define IS_LOOPBACK(ill) \
        ((ill)->ill_phyint->phyint_flags & PHYI_LOOPBACK)

/* Is this an IPMP meta-interface ILL? */
#define IS_IPMP(ill)                                                    \
        ((ill)->ill_phyint->phyint_flags & PHYI_IPMP)

/* Is this ILL under an IPMP meta-interface? (aka "in a group?") */
#define IS_UNDER_IPMP(ill)                                              \
        ((ill)->ill_grp != NULL && !IS_IPMP(ill))

/* Is ill1 in the same illgrp as ill2? */
#define IS_IN_SAME_ILLGRP(ill1, ill2)                                   \
        ((ill1)->ill_grp != NULL && ((ill1)->ill_grp == (ill2)->ill_grp))

/* Is ill1 on the same LAN as ill2? */
#define IS_ON_SAME_LAN(ill1, ill2)                                      \
        ((ill1) == (ill2) || IS_IN_SAME_ILLGRP(ill1, ill2))

#define ILL_OTHER(ill)                                                  \
        ((ill)->ill_isv6 ? (ill)->ill_phyint->phyint_illv4 :            \
            (ill)->ill_phyint->phyint_illv6)

/*
 * IPMP group ILL state structure -- up to two per IPMP group (V4 and V6).
 * Created when the V4 and/or V6 IPMP meta-interface is I_PLINK'd.  It is
 * guaranteed to persist while there are interfaces of that type in the group.
 * In general, most fields are accessed outside of the IPSQ (e.g., in the
 * datapath), and thus use locks in addition to the IPSQ for protection.
 *
 * synchronization:             read                    write
 *
 *      ig_if                   ipsq or ill_g_lock      ipsq and ill_g_lock
 *      ig_actif                ipsq or ipmp_lock       ipsq and ipmp_lock
 *      ig_nactif               ipsq or ipmp_lock       ipsq and ipmp_lock
 *      ig_next_ill             ipsq or ipmp_lock       ipsq and ipmp_lock
 *      ig_ipmp_ill             write once              write once
 *      ig_cast_ill             ipsq or ipmp_lock       ipsq and ipmp_lock
```



```
 *      ig_arpent               ipsq                    ipsq
 *      ig_mtu                  ipsq                    ipsq
 *      ig_mc_mtu               ipsq                    ipsq
 */
typedef struct ipmp_illgrp_s {
        list_t          ig_if;          /* list of all interfaces */
        list_t          ig_actif;       /* list of active interfaces */
        uint_t          ig_nactif;      /* number of active interfaces */
        struct ill_s    *ig_next_ill;   /* next active interface to use */
        struct ill_s    *ig_ipmp_ill;   /* backpointer to IPMP meta-interface */
        struct ill_s    *ig_cast_ill;   /* nominated ill for multi/broadcast */
        list_t          ig_arpent;      /* list of ARP entries */
        uint_t          ig_mtu;         /* ig_ipmp_ill->ill_mtu */
        uint_t          ig_mc_mtu;      /* ig_ipmp_ill->ill_mc_mtu */
} ipmp_illgrp_t;

/*
 * IPMP group state structure -- one per IPMP group.  Created when the
 * IPMP meta-interface is plumbed; it is guaranteed to persist while there
 * are interfaces in it.
 *
 * ipmp_grp_t synchronization:          read                    write
 *
 *      gr_name                         ipmp_lock               ipmp_lock
 *      gr_ifname                       write once              write once
 *      gr_mactype                      ipmp_lock               ipmp_lock
 *      gr_phyint                       write once              write once
 *      gr_nif                          ipmp_lock               ipmp_lock
 *      gr_nactif                       ipsq                    ipsq
 *      gr_v4                           ipmp_lock               ipmp_lock
 *      gr_v6                           ipmp_lock               ipmp_lock
 *      gr_nv4                          ipmp_lock               ipmp_lock
 *      gr_nv6                          ipmp_lock               ipmp_lock
 *      gr_pendv4                       ipmp_lock               ipmp_lock
 *      gr_pendv6                       ipmp_lock               ipmp_lock
 *      gr_linkdownmp                   ipsq                    ipsq
 *      gr_ksp                          ipmp_lock               ipmp_lock
 *      gr_kstats0                      atomic                  atomic
 */
typedef struct ipmp_grp_s {
        char            gr_name[LIFGRNAMSIZ];   /* group name */
        char            gr_ifname[LIFNAMSIZ];   /* interface name */
        t_uscalar_t     gr_mactype;     /* DLPI mactype of group */
        phyint_t        *gr_phyint;     /* IPMP group phyint */
        uint_t          gr_nif;         /* number of interfaces in group */
        uint_t          gr_nactif;      /* number of active interfaces */
        ipmp_illgrp_t   *gr_v4;         /* V4 group information */
        ipmp_illgrp_t   *gr_v6;         /* V6 group information */
        uint_t          gr_nv4;         /* number of ills in V4 group */
        uint_t          gr_nv6;         /* number of ills in V6 group */
        uint_t          gr_pendv4;      /* number of pending ills in V4 group */
        uint_t          gr_pendv6;      /* number of pending ills in V6 group */
        mblk_t          *gr_linkdownmp; /* message used to bring link down */
        kstat_t         *gr_ksp;        /* group kstat pointer */
        uint64_t        gr_kstats0[IPMP_KSTAT_MAX]; /* baseline group kstats */
} ipmp_grp_t;

/*
 * IPMP ARP entry -- one per SIOCS*ARP entry tied to the group.  Used to keep
 * ARP up-to-date as the active set of interfaces in the group changes.
 */
typedef struct ipmp_arpent_s {
        ipaddr_t        ia_ipaddr;      /* IP address for this entry */
        boolean_t       ia_proxyarp;    /* proxy ARP entry? */
        boolean_t       ia_notified;    /* ARP notified about this entry? */
        list_node_t     ia_node;        /* next ARP entry in list */
```

```
1580         uint16_t        ia_flags;       /* nce_flags for the address */
1581         size_t          ia_lladdr_len;
1582         uchar_t         *ia_lladdr;
1583 } ipmp_arpent_t;

1585 struct arl_s;

1587 /*
1588  * Per-ill capabilities.
1589  */
1590 struct ill_hcksum_capab_s {
1591         uint_t  ill_hcksum_version;     /* interface version */
1592         uint_t  ill_hcksum_txflags;     /* capabilities on transmit */
1593 };

1595 struct ill_zerocopy_capab_s {
1596         uint_t  ill_zerocopy_version;   /* interface version */
1597         uint_t  ill_zerocopy_flags;     /* capabilities */
1598 };

1600 struct ill_lso_capab_s {
1601         uint_t  ill_lso_flags;          /* capabilities */
1602         uint_t  ill_lso_max;            /* maximum size of payload */
1603 };

1605 /*
1606  * IP Lower level Structure.
1607  * Instance data structure in ip_open when there is a device below us.
1608  */
1609 typedef struct ill_s {
1610         pfillinput_t ill_inputfn;       /* Fast input function selector */
1611         ill_if_t *ill_ifptr;            /* pointer to interface type */
1612         queue_t *ill_rq;                /* Read queue. */
1613         queue_t *ill_wq;                /* Write queue. */

1615         int     ill_error;              /* Error value sent up by device. */

1617         ipif_t  *ill_ipif;              /* Interface chain for this ILL. */

1619         uint_t  ill_ipif_up_count;      /* Number of IPIFs currently up. */
1620         uint_t  ill_max_frag;           /* Max IDU from DLPI. */
1621         uint_t  ill_current_frag;       /* Current IDU from DLPI. */
1622         uint_t  ill_mtu;                /* User-specified MTU; SIOCSLIFMTU */
1623         uint_t  ill_mc_mtu;             /* MTU for multi/broadcast */
1624         uint_t  ill_metric;             /* BSD if metric, for compatibility. */
1625         char    *ill_name;              /* Our name. */
1626         uint_t  ill_ipif_dup_count;     /* Number of duplicate addresses. */
1627         uint_t  ill_name_length;        /* Name length, incl. terminator. */
1628         uint_t  ill_net_type;           /* IRE_IF_RESOLVER/IRE_IF_NORESOLVER. */
1629         /*
1630          * Physical Point of Attachment num.  If DLPI style 1 provider
1631          * then this is derived from the devname.
1632          */
1633         uint_t  ill_ppa;
1634         t_uscalar_t     ill_sap;
1635         t_scalar_t      ill_sap_length; /* Including sign (for position) */
1636         uint_t  ill_phys_addr_length;   /* Excluding the sap. */
1637         uint_t  ill_bcast_addr_length;  /* Only set when the DL provider */
1638                                         /* supports broadcast. */
1639         t_uscalar_t     ill_mactype;
1640         uint8_t *ill_frag_ptr;          /* Reassembly state. */
1641         timeout_id_t ill_frag_timer_id; /* timeout id for the frag timer */
1642         ipfb_t  *ill_frag_hash_tbl;     /* Fragment hash list head. */

1644         krwlock_t ill_mcast_lock;       /* Protects multicast state */
1645         kmutex_t ill_mcast_serializer;  /* Serialize across ilg and ilm state */
```



```
1646         ilm_t   *ill_ilm;               /* Multicast membership for ill */
1647         uint_t  ill_global_timer;       /* for IGMPv3/MLDv2 general queries */
1648         int     ill_mcast_type;         /* type of router which is querier */
1649                                         /* on this interface */
1650         uint16_t ill_mcast_v1_time;     /* # slow timeouts since last v1 qry */
1651         uint16_t ill_mcast_v2_time;     /* # slow timeouts since last v2 qry */
1652         uint8_t ill_mcast_v1_tset;      /* 1 => timer is set; 0 => not set */
1653         uint8_t ill_mcast_v2_tset;      /* 1 => timer is set; 0 => not set */

1655         uint8_t ill_mcast_rv;           /* IGMPv3/MLDv2 robustness variable */
1656         int     ill_mcast_qi;           /* IGMPv3/MLDv2 query interval var */

1658         /*
1659          * All non-NULL cells between 'ill_first_mp_to_free' and
1660          * 'ill_last_mp_to_free' are freed in ill_delete.
1661          */
1662 #define ill_first_mp_to_free    ill_bcast_mp
1663         mblk_t  *ill_bcast_mp;          /* DLPI header for broadcasts. */
1664         mblk_t  *ill_unbind_mp;         /* unbind mp from ill_dl_up() */
1665         mblk_t  *ill_promiscoff_mp;     /* for ill_leave_allmulti() */
1666         mblk_t  *ill_dlpi_deferred;     /* b_next chain of control messages */
1667         mblk_t  *ill_dest_addr_mp;      /* mblk which holds ill_dest_addr */
1668         mblk_t  *ill_replumb_mp;        /* replumb mp from ill_replumb() */
1669         mblk_t  *ill_phys_addr_mp;      /* mblk which holds ill_phys_addr */
1670         mblk_t  *ill_mcast_deferred;    /* b_next chain of IGMP/MLD packets */
1671 #define ill_last_mp_to_free     ill_mcast_deferred

1673         cred_t  *ill_credp;             /* opener's credentials */
1674         uint8_t *ill_phys_addr;         /* ill_phys_addr_mp->b_rptr + off */
1675         uint8_t *ill_dest_addr;         /* ill_dest_addr_mp->b_rptr + off */

1677         uint_t  ill_state_flags;        /* see ILL_* flags above */

1679         /* Following bit fields protected by ipsq_t */
1680         uint_t
1681                 ill_needs_attach : 1,
1682                 ill_reserved : 1,
1683                 ill_isv6 : 1,
1684                 ill_dlpi_style_set : 1,

1686                 ill_ifname_pending : 1,
1687                 ill_logical_down : 1,
1688                 ill_dl_up : 1,
1689                 ill_up_ipifs : 1,

1691                 ill_note_link : 1,      /* supports link-up notification */
1692                 ill_capab_reneg : 1, /* capability renegotiation to be done */
1693                 ill_dld_capab_inprog : 1, /* direct dld capab call in prog */
1694                 ill_need_recover_multicast : 1,

1696                 ill_replumbing : 1,
1697                 ill_arl_dlpi_pending : 1,
1698                 ill_grp_pending : 1,

1700                 ill_pad_to_bit_31 : 17;

1702         /* Following bit fields protected by ill_lock */
1703         uint_t
1704                 ill_fragtimer_executing : 1,
1705                 ill_fragtimer_needrestart : 1,
1706                 ill_manual_token : 1,   /* system won't override ill_token */
1707                 /*
1708                  * ill_manual_linklocal : system will not change the
1709                  * linklocal whenever ill_token changes.
1710                  */
1711                 ill_manual_linklocal : 1,
```

```
1713                ill_manual_dst_linklocal : 1, /* same for pt-pt dst linklocal */

1715                ill_pad_bit_31 : 27;

1717        /*
1718         * Used in SIOCSIFMUXID and SIOCGIFMUXID for 'ifconfig unplumb'.
1719         */
1720        int     ill_muxid;              /* muxid returned from plink */

1722        /* Used for IP frag reassembly throttling on a per ILL basis.  */
1723        uint_t  ill_ipf_gen;            /* Generation of next fragment queue */
1724        uint_t  ill_frag_count;         /* Count of all reassembly mblk bytes */
1725        uint_t  ill_frag_free_num_pkts;  /* num of fragmented packets to free */
1726        clock_t ill_last_frag_clean_time; /* time when frag's were pruned */
1727        int     ill_type;               /* From <net/if_types.h> */
1728        uint_t  ill_dlpi_multicast_state;       /* See below IDS_* */
1729        uint_t  ill_dlpi_fastpath_state;        /* See below IDS_* */

1731        /*
1732         * Capabilities related fields.
1733         */
1734        uint_t  ill_dlpi_capab_state;   /* State of capability query, IDCS_* */
1735        uint_t  ill_capab_pending_cnt;
1736        uint64_t ill_capabilities;      /* Enabled capabilities, ILL_CAPAB_* */
1737        ill_hcksum_capab_t *ill_hcksum_capab; /* H/W cksumming capabilities */
1738        ill_zerocopy_capab_t *ill_zerocopy_capab; /* Zero-copy capabilities */
1739        ill_dld_capab_t *ill_dld_capab; /* DLD capabilities */
1740        ill_lso_capab_t *ill_lso_capab; /* Large Segment Offload capabilities */
1741        mblk_t  *ill_capab_reset_mp;    /* Preallocated mblk for capab reset */

1743        uint8_t ill_max_hops;   /* Maximum hops for any logical interface */
1744        uint_t  ill_user_mtu;   /* User-specified MTU via SIOCSLIFLNKINFO */
1745        uint32_t ill_reachable_time;    /* Value for ND algorithm in msec */
1746        uint32_t ill_reachable_retrans_time; /* Value for ND algorithm msec */
1747        uint_t  ill_max_buf;            /* Max # of req to buffer for ND */
1748        in6_addr_t      ill_token;      /* IPv6 interface id */
1749        in6_addr_t      ill_dest_token; /* Destination IPv6 interface id */
1750        uint_t          ill_token_length;
1751        uint32_t        ill_xmit_count;         /* ndp max multicast xmits */
1752        mib2_ipIfStatsEntry_t   *ill_ip_mib;    /* ver indep. interface mib */
1753        mib2_ipv6IfIcmpEntry_t  *ill_icmp6_mib; /* Per interface mib */

1755        phyint_t                *ill_phyint;
1756        uint64_t                ill_flags;

1758        kmutex_t        ill_lock;       /* Please see table below */
1759        /*
1760         * The ill_nd_lla* fields handle the link layer address option
1761         * from neighbor discovery. This is used for external IPv6
1762         * address resolution.
1763         */
1764        mblk_t          *ill_nd_lla_mp; /* mblk which holds ill_nd_lla */
1765        uint8_t         *ill_nd_lla;    /* Link Layer Address */
1766        uint_t          ill_nd_lla_len; /* Link Layer Address length */
1767        /*
1768         * We have 4 phys_addr_req's sent down. This field keeps track
1769         * of which one is pending.
1770         */
1771        t_uscalar_t     ill_phys_addr_pend; /* which dl_phys_addr_req pending */
1772        /*
1773         * Used to save errors that occur during plumbing
1774         */
1775        uint_t          ill_ifname_pending_err;
1776        avl_node_t      ill_avl_byppa; /* avl node based on ppa */
1777        list_t          ill_nce; /* pointer to nce_s list */
```



```
1778        uint_t          ill_refcnt;     /* active refcnt by threads */
1779        uint_t          ill_ire_cnt;    /* ires associated with this ill */
1780        kcondvar_t      ill_cv;
1781        uint_t          ill_ncec_cnt;   /* ncecs associated with this ill */
1782        uint_t          ill_nce_cnt;    /* nces associated with this ill */
1783        uint_t          ill_waiters;    /* threads waiting in ipsq_enter */
1784        /*
1785         * Contains the upper read queue pointer of the module immediately
1786         * beneath IP.  This field allows IP to validate sub-capability
1787         * acknowledgments coming up from downstream.
1788         */
1789        queue_t         *ill_lmod_rq;   /* read queue pointer of module below */
1790        uint_t          ill_lmod_cnt;   /* number of modules beneath IP */
1791        ip_m_t          *ill_media;     /* media specific params/functions */
1792        t_uscalar_t     ill_dlpi_pending; /* Last DLPI primitive issued */
1793        uint_t          ill_usesrc_ifindex; /* use src addr from this ILL */
1794        struct ill_s    *ill_usesrc_grp_next; /* Next ILL in the usesrc group */
1795        boolean_t       ill_trace_disable;      /* True when alloc fails */
1796        zoneid_t        ill_zoneid;
1797        ip_stack_t      *ill_ipst;      /* Corresponds to a netstack_hold */
1798        uint32_t        ill_dhcpinit;   /* IP_DHCPINIT_IFs for ill */
1799        void            *ill_flownotify_mh; /* Tx flow ctl, mac cb handle */
1800        uint_t          ill_ilm_cnt;    /* ilms referencing this ill */
1801        uint_t          ill_ipallmulti_cnt; /* ip_join_allmulti() calls */
1802        ilm_t           *ill_ipallmulti_ilm;

1804        mblk_t          *ill_saved_ire_mp; /* Allocated for each extra IRE */
1805                                        /* with ire_ill set so they can */
1806                                        /* survive the ill going down and up. */
1807        kmutex_t        ill_saved_ire_lock; /* Protects ill_saved_ire_mp, cnt */
1808        uint_t          ill_saved_ire_cnt;      /* # entries */
1809        struct arl_ill_common_s    *ill_common;
1810        ire_t           *ill_ire_multicast; /* IRE_MULTICAST for ill */
1811        clock_t         ill_defend_start;   /* start of 1 hour period */
1812        uint_t          ill_defend_count;   /* # of announce/defends per ill */
1813        /*
1814         * IPMP fields.
1815         */
1816        ipmp_illgrp_t   *ill_grp;       /* IPMP group information */
1817        list_node_t     ill_actnode;    /* next active ill in group */
1818        list_node_t     ill_grpnode;    /* next ill in group */
1819        ipif_t          *ill_src_ipif;  /* source address selection rotor */
1820        ipif_t          *ill_move_ipif; /* ipif awaiting move to new ill */
1821        boolean_t       ill_nom_cast;   /* nominated for mcast/bcast */
1822        uint_t          ill_bound_cnt;  /* # of data addresses bound to ill */
1823        ipif_t          *ill_bound_ipif; /* ipif chain bound to ill */
1824        timeout_id_t    ill_refresh_tid; /* ill refresh retry timeout id */

1826        uint32_t        ill_mrouter_cnt; /* mrouter allmulti joins */
1827        uint32_t        ill_allowed_ips_cnt;
1828        in6_addr_t      *ill_allowed_ips;

1830        /* list of multicast physical addresses joined on this ill */
1831        multiphysaddr_t *ill_mphysaddr_list;
1832 } ill_t;

1834 /*
1835  * ILL_FREE_OK() means that there are no incoming pointer references
1836  * to the ill.
1837  */
1838 #define ILL_FREE_OK(ill)                                                \
1839        ((ill)->ill_ire_cnt == 0 && (ill)->ill_ilm_cnt == 0 &&  \
1840        (ill)->ill_ncec_cnt == 0 && (ill)->ill_nce_cnt == 0)

1842 /*
1843  * An ipif/ill can be marked down only when the ire and ncec references
```

```
 * to that ipif/ill goes to zero. ILL_DOWN_OK() is a necessary condition
 * quiescence checks. See comments above IPIF_DOWN_OK for details
 * on why ires and nces are selectively considered for this macro.
 */
#define ILL_DOWN_OK(ill)                                        \
        (ill->ill_ire_cnt == 0 && ill->ill_ncec_cnt == 0 &&     \
        ill->ill_nce_cnt == 0)

/*
 * The following table lists the protection levels of the various members
 * of the ill_t. Same notation as that used for ipif_t above is used.
 *
 *                              Write                   Read
 *
 * ill_ifptr                    ill_g_lock + s          Write once
 * ill_rq                       ipsq                    Write once
 * ill_wq                       ipsq                    Write once
 *
 * ill_error                    ipsq                    None
 * ill_ipif                     ill_g_lock + ipsq       ill_g_lock OR ipsq
 * ill_ipif_up_count            ill_lock + ipsq         ill_lock OR ipsq
 * ill_max_frag                 ill_lock                ill_lock
 * ill_current_frag             ill_lock                ill_lock
 *
 * ill_name                     ill_g_lock + ipsq       Write once
 * ill_name_length              ill_g_lock + ipsq       Write once
 * ill_ndd_name                 ipsq                    Write once
 * ill_net_type                 ipsq                    Write once
 * ill_ppa                      ill_g_lock + ipsq       Write once
 * ill_sap                      ipsq + down ill         Write once
 * ill_sap_length               ipsq + down ill         Write once
 * ill_phys_addr_length         ipsq + down ill         Write once
 *
 * ill_bcast_addr_length        ipsq                    ipsq
 * ill_mactype                  ipsq                    ipsq
 * ill_frag_ptr                 ipsq                    ipsq
 *
 * ill_frag_timer_id            ill_lock                ill_lock
 * ill_frag_hash_tbl            ipsq                    up ill
 * ill_ilm                      ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_global_timer             ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_mcast_type               ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_mcast_v1_time            ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_mcast_v2_time            ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_mcast_v1_tset            ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_mcast_v2_tset            ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_mcast_rv                 ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 * ill_mcast_qi                 ill_mcast_lock(WRITER)  ill_mcast_lock(READER)
 *
 * ill_down_mp                  ipsq                    ipsq
 * ill_dlpi_deferred            ill_lock                ill_lock
 * ill_dlpi_pending             ipsq + ill_lock         ipsq or ill_lock or
 *                                                      absence of ipsq writer.
 * ill_phys_addr_mp             ipsq + down ill         only when ill is up
 * ill_mcast_deferred           ill_lock                ill_lock
 * ill_phys_addr                ipsq + down ill         only when ill is up
 * ill_dest_addr_mp             ipsq + down ill         only when ill is up
 * ill_dest_addr                ipsq + down ill         only when ill is up
 *
 * ill_state_flags              ill_lock                ill_lock
 * exclusive bit flags          ipsq_t                  ipsq_t
 * shared bit flags             ill_lock                ill_lock
 *
 * ill_muxid                    ipsq                    Not atomic
 *
 * ill_ipf_gen                  Not atomic
```



```
 * ill_frag_count               atomics                 atomics
 * ill_type                     ipsq + down ill         only when ill is up
 * ill_dlpi_multicast_state     ill_lock                ill_lock
 * ill_dlpi_fastpath_state      ill_lock                ill_lock
 * ill_dlpi_capab_state         ipsq                    ipsq
 * ill_max_hops                 ipsq                    Not atomic
 *
 * ill_mtu                      ill_lock                None
 * ill_mc_mtu                   ill_lock                None
 *
 * ill_user_mtu                 ipsq + ill_lock         ill_lock
 * ill_reachable_time           ipsq + ill_lock         ill_lock
 * ill_reachable_retrans_time   ipsq + ill_lock         ill_lock
 * ill_max_buf                  ipsq + ill_lock         ill_lock
 *
 * Next 2 fields need ill_lock because of the get ioctls. They should not
 * report partially updated results without executing in the ipsq.
 * ill_token                    ipsq + ill_lock         ill_lock
 * ill_token_length             ipsq + ill_lock         ill_lock
 * ill_dest_token               ipsq + down ill         only when ill is up
 * ill_xmit_count               ipsq + down ill         write once
 * ill_ip6_mib                  ipsq + down ill         only when ill is up
 * ill_icmp6_mib                ipsq + down ill         only when ill is up
 *
 * ill_phyint                   ipsq, ill_g_lock, ill_lock      Any of them
 * ill_flags                    ill_lock                ill_lock
 * ill_nd_lla_mp                ipsq + down ill         only when ill is up
 * ill_nd_lla                   ipsq + down ill         only when ill is up
 * ill_nd_lla_len               ipsq + down ill         only when ill is up
 * ill_phys_addr_pend           ipsq + down ill         only when ill is up
 * ill_ifname_pending_err       ipsq                    ipsq
 * ill_avl_byppa                ipsq, ill_g_lock        write once
 *
 * ill_fastpath_list            ill_lock                ill_lock
 * ill_refcnt                   ill_lock                ill_lock
 * ill_ire_cnt                  ill_lock                ill_lock
 * ill_cv                       ill_lock                ill_lock
 * ill_ncec_cnt                 ill_lock                ill_lock
 * ill_nce_cnt                  ill_lock                ill_lock
 * ill_ilm_cnt                  ill_lock                ill_lock
 * ill_src_ipif                 ill_g_lock              ill_g_lock
 * ill_trace                    ill_lock                ill_lock
 * ill_usesrc_grp_next          ill_g_usesrc_lock       ill_g_usesrc_lock
 * ill_dhcpinit                 atomics                 atomics
 * ill_flownotify_mh            write once              write once
 * ill_capab_pending_cnt        ipsq                    ipsq
 * ill_ipallmulti_cnt           ill_lock                ill_lock
 * ill_ipallmulti_ilm           ill_lock                ill_lock
 * ill_saved_ire_mp             ill_saved_ire_lock      ill_saved_ire_lock
 * ill_saved_ire_cnt            ill_saved_ire_lock      ill_saved_ire_lock
 * ill_arl                      ???                     ???
 * ill_ire_multicast            ipsq + quiescent        none
 * ill_bound_ipif               ipsq                    ipsq
 * ill_actnode                  ipsq + ipmp_lock        ipsq OR ipmp_lock
 * ill_grpnode                  ipsq + ill_g_lock       ipsq OR ill_g_lock
 * ill_src_ipif                 ill_g_lock              ill_g_lock
 * ill_move_ipif                ipsq                    ipsq
 * ill_nom_cast                 ipsq                    ipsq OR advisory
 * ill_refresh_tid              ill_lock                ill_lock
 * ill_grp (for IPMP ill)       write once              write once
 * ill_grp (for underlying ill) ipsq + ill_g_lock       ipsq OR ill_g_lock
 * ill_grp_pending              ill_mcast_serializer    ill_mcast_serializer
 * ill_mrouter_cnt              atomics                 atomics
 * ill_mphysaddr_list   ill_lock                ill_lock
 *
 * NOTE: It's OK to make heuristic decisions on an underlying interface
```

```
 *       by using IS_UNDER_IPMP() or comparing ill_grp's raw pointer value.
 */

/*
 * For ioctl restart mechanism see ip_reprocess_ioctl()
 */
struct ip_ioctl_cmd_s;

typedef int (*ifunc_t)(ipif_t *, struct sockaddr_in *, queue_t *, mblk_t *,
    struct ip_ioctl_cmd_s *, void *);

typedef struct ip_ioctl_cmd_s {
        int     ipi_cmd;
        size_t  ipi_copyin_size;
        uint_t  ipi_flags;
        uint_t  ipi_cmd_type;
        ifunc_t ipi_func;
        ifunc_t ipi_func_restart;
} ip_ioctl_cmd_t;

/*
 * ipi_cmd_type:
 *
 * IF_CMD               1       old style ifreq cmd
 * LIF_CMD              2       new style lifreq cmd
 * ARP_CMD              3       arpreq cmd
 * XARP_CMD             4       xarpreq cmd
 * MSFILT_CMD           5       multicast source filter cmd
 * MISC_CMD             6       misc cmd (not a more specific one above)
 */

enum { IF_CMD = 1, LIF_CMD, ARP_CMD, XARP_CMD, MSFILT_CMD, MISC_CMD };

#define IPI_DONTCARE    0       /* For ioctl encoded values that don't matter */

/* Flag values in ipi_flags */
#define IPI_PRIV        0x1     /* Root only command */
#define IPI_MODOK       0x2     /* Permitted on mod instance of IP */
#define IPI_WR          0x4     /* Need to grab writer access */
#define IPI_GET_CMD     0x8     /* branch to mi_copyout on success */
/*      unused          0x10    */
#define IPI_NULL_BCONT  0x20    /* ioctl has not data and hence no b_cont */

extern ip_ioctl_cmd_t   ip_ndx_ioctl_table[];
extern ip_ioctl_cmd_t   ip_misc_ioctl_table[];
extern int ip_ndx_ioctl_count;
extern int ip_misc_ioctl_count;

/* Passed down by ARP to IP during I_PLINK/I_PUNLINK */
typedef struct ipmx_s {
        char    ipmx_name[LIFNAMSIZ];           /* if name */
        uint_t
                ipmx_arpdev_stream : 1,         /* This is the arp stream */
                ipmx_notused : 31;
} ipmx_t;

/*
 * State for detecting if a driver supports certain features.
 * Support for DL_ENABMULTI_REQ uses ill_dlpi_multicast_state.
 * Support for DLPI M_DATA fastpath uses ill_dlpi_fastpath_state.
 */
#define IDS_UNKNOWN     0       /* No DLPI request sent */
#define IDS_INPROGRESS  1       /* DLPI request sent */
#define IDS_OK          2       /* DLPI request completed successfully */
#define IDS_FAILED      3       /* DLPI request failed */
```



```
/* Support for DL_CAPABILITY_REQ uses ill_dlpi_capab_state. */
enum {
        IDCS_UNKNOWN,
        IDCS_PROBE_SENT,
        IDCS_OK,
        IDCS_RESET_SENT,
        IDCS_RENEG,
        IDCS_FAILED
};

/* Extended NDP Management Structure */
typedef struct ipndp_s {
        ndgetf_t        ip_ndp_getf;
        ndsetf_t        ip_ndp_setf;
        caddr_t         ip_ndp_data;
        char            *ip_ndp_name;
} ipndp_t;

/* IXA Notification types */
typedef enum {
        IXAN_LSO,       /* LSO capability change */
        IXAN_PMTU,      /* PMTU change */
        IXAN_ZCOPY      /* ZEROCOPY capability change */
} ixa_notify_type_t;

typedef uint_t ixa_notify_arg_t;

typedef void    (*ixa_notify_t)(void *, ip_xmit_attr_t *ixa, ixa_notify_type_t,
    ixa_notify_arg_t);

/*
 * Attribute flags that are common to the transmit and receive attributes
 */
#define IAF_IS_IPV4             0x80000000      /* ipsec_*_v4 */
#define IAF_TRUSTED_ICMP        0x40000000      /* ipsec_*_icmp_loopback */
#define IAF_NO_LOOP_ZONEID_SET  0x20000000      /* Zone that shouldn't have */
                                                /* a copy */
#define IAF_LOOPBACK_COPY       0x10000000      /* For multi and broadcast */

#define IAF_MASK                0xf0000000      /* Flags that are common */

/*
 * Transmit side attributes used between the transport protocols and IP as
 * well as inside IP. It is also used to cache information in the conn_t i.e.
 * replaces conn_ire and the IPsec caching in the conn_t.
 */
struct ip_xmit_attr_s {
        iaflags_t       ixa_flags;      /* IXAF_*. See below */

        uint32_t        ixa_free_flags; /* IXA_FREE_*. See below */
        uint32_t        ixa_refcnt;     /* Using atomics */

        /*
         * Always initialized independently of ixa_flags settings.
         * Used by ip_xmit so we keep them up front for cache locality.
         */
        uint32_t        ixa_xmit_hint;  /* For ECMP and GLD TX ring fanout */
        uint_t          ixa_pktlen;     /* Always set. For frag and stats */
        zoneid_t        ixa_zoneid;     /* Assumed always set */

        /* Always set for conn_ip_output(); might be stale */
        /*
         * Since TCP keeps the conn_t around past the process going away
         * we need to use the "notr" (e.g, ire_refhold_notr) for ixa_ire,
         * ixa_nce, and ixa_dce.
         */
```

```
2108         ire_t           *ixa_ire;       /* Forwarding table entry */
2109         uint_t          ixa_ire_generation;
2110         nce_t           *ixa_nce;       /* Neighbor cache entry */
2111         dce_t           *ixa_dce;       /* Destination cache entry */
2112         uint_t          ixa_dce_generation;
2113         uint_t          ixa_src_generation;     /* If IXAF_VERIFY_SOURCE */

2115         uint32_t        ixa_src_preferences;    /* prefs for src addr select */
2116         uint32_t        ixa_pmtu;               /* IXAF_VERIFY_PMTU */

2118         /* Set by ULP if IXAF_VERIFY_PMTU; otherwise set by IP */
2119         uint32_t        ixa_fragsize;

2121         int8_t          ixa_use_min_mtu;        /* IXAF_USE_MIN_MTU values */

2123         pfirepostfrag_t ixa_postfragfn;         /* Set internally in IP */

2125         in6_addr_t      ixa_nexthop_v6;         /* IXAF_NEXTHOP_SET */
2126 #define ixa_nexthop_v4  V4_PART_OF_V6(ixa_nexthop_v6)

2128         zoneid_t        ixa_no_loop_zoneid;     /* IXAF_NO_LOOP_ZONEID_SET */

2130         uint_t          ixa_scopeid;            /* For IPv6 link-locals */

2132         uint_t          ixa_broadcast_ttl;      /* IXAF_BROACAST_TTL_SET */

2134         uint_t          ixa_multicast_ttl;      /* Assumed set for multicast */
2135         uint_t          ixa_multicast_ifindex;  /* Assumed set for multicast */
2136         ipaddr_t        ixa_multicast_ifaddr;   /* Assumed set for multicast */

2138         int             ixa_raw_cksum_offset;   /* If IXAF_SET_RAW_CKSUM */

2140         uint32_t        ixa_ident;              /* For IPv6 fragment header */

2142         uint64_t        ixa_conn_id;            /* Used by DTrace */
2143         /*
2144          * Cached LSO information.
2145          */
2146         ill_lso_capab_t ixa_lso_capab;          /* Valid when IXAF_LSO_CAPAB */

2148         uint64_t        ixa_ipsec_policy_gen;   /* Generation from iph_gen */
2149         /*
2150          * The following IPsec fields are only initialized when
2151          * IXAF_IPSEC_SECURE is set. Otherwise they contain garbage.
2152          */
2153         ipsec_latch_t   *ixa_ipsec_latch;       /* Just the ids */
2154         struct ipsa_s   *ixa_ipsec_ah_sa;       /* Hard reference SA for AH */
2155         struct ipsa_s   *ixa_ipsec_esp_sa;      /* Hard reference SA for ESP */
2156         struct ipsec_policy_s   *ixa_ipsec_policy; /* why are we here? */
2157         struct ipsec_action_s   *ixa_ipsec_action; /* For reflected packets */
2158         ipsa_ref_t      ixa_ipsec_ref[2];       /* Soft reference to SA */
2159                                                 /* 0: ESP, 1: AH */

2161         /*
2162          * The selectors here are potentially different than the SPD rule's
2163          * selectors, and we need to have both available for IKEv2.
2164          *
2165          * NOTE: "Source" and "Dest" are w.r.t. outbound datagrams.  Ports can
2166          *       be zero, and the protocol number is needed to make the ports
2167          *       significant.
2168          */
2169         uint16_t ixa_ipsec_src_port;    /* Source port number of d-gram. */
2170         uint16_t ixa_ipsec_dst_port;    /* Destination port number of d-gram. */
2171         uint8_t  ixa_ipsec_icmp_type;   /* ICMP type of d-gram */
2172         uint8_t  ixa_ipsec_icmp_code;   /* ICMP code of d-gram */
```



```
2174         sa_family_t ixa_ipsec_inaf;     /* Inner address family */
2175 #define IXA_MAX_ADDRLEN 4       /* Max addr len. (in 32-bit words) */
2176         uint32_t ixa_ipsec_insrc[IXA_MAX_ADDRLEN];      /* Inner src address */
2177         uint32_t ixa_ipsec_indst[IXA_MAX_ADDRLEN];      /* Inner dest address */
2178         uint8_t  ixa_ipsec_insrcpfx;    /* Inner source prefix */
2179         uint8_t  ixa_ipsec_indstpfx;    /* Inner destination prefix */

2181         uint8_t ixa_ipsec_proto;        /* IP protocol number for d-gram. */

2183         /* Always initialized independently of ixa_flags settings */
2184         uint_t          ixa_ifindex;    /* Assumed always set */
2185         uint16_t        ixa_ip_hdr_length; /* Points to ULP header */
2186         uint8_t         ixa_protocol;   /* Protocol number for ULP cksum */
2187         ts_label_t      *ixa_tsl;       /* Always set. NULL if not TX */
2188         ip_stack_t      *ixa_ipst;      /* Always set */
2189         uint32_t        ixa_extra_ident; /* Set if LSO */
2190         cred_t          *ixa_cred;      /* For getpeerucred */
2191         pid_t           ixa_cpid;       /* For getpeerucred */

2193 #ifdef DEBUG
2194         kthread_t       *ixa_curthread; /* For serialization assert */
2195 #endif
2196         squeue_t        *ixa_sqp;       /* Set from conn_sqp as a hint */
2197         uintptr_t       ixa_cookie;     /* cookie to use for tx flow control */

2199         /*
2200          * Must be set by ULP if any of IXAF_VERIFY_LSO, IXAF_VERIFY_PMTU,
2201          * or IXAF_VERIFY_ZCOPY is set.
2202          */
2203         ixa_notify_t    ixa_notify;     /* Registered upcall notify function */
2204         void            *ixa_notify_cookie; /* ULP cookie for ixa_notify */
2205 };

2207 /*
2208  * Flags to indicate which transmit attributes are set.
2209  * Split into "xxx_SET" ones which indicate that the "xxx" field it set, and
2210  * single flags.
2211  */
2212 #define IXAF_REACH_CONF         0x00000001      /* Reachability confirmation */
2213 #define IXAF_BROADCAST_TTL_SET  0x00000002      /* ixa_broadcast_ttl valid */
2214 #define IXAF_SET_SOURCE         0x00000004      /* Replace if broadcast */
2215 #define IXAF_USE_MIN_MTU        0x00000008      /* IPV6_USE_MIN_MTU */

2217 #define IXAF_DONTFRAG           0x00000010      /* IP*_DONTFRAG */
2218 #define IXAF_VERIFY_PMTU        0x00000020      /* ixa_pmtu/ixa_fragsize set */
2219 #define IXAF_PMTU_DISCOVERY     0x00000040      /* Create/use PMTU state */
2220 #define IXAF_MULTICAST_LOOP     0x00000080      /* IP_MULTICAST_LOOP */

2222 #define IXAF_IPSEC_SECURE       0x00000100      /* Need IPsec processing */
2223 #define IXAF_UCRED_TSL          0x00000200      /* ixa_tsl from SCM_UCRED */
2224 #define IXAF_DONTROUTE          0x00000400      /* SO_DONTROUTE */
2225 #define IXAF_NO_IPSEC           0x00000800      /* Ignore policy */

2227 #define IXAF_PMTU_TOO_SMALL     0x00001000      /* PMTU too small */
2228 #define IXAF_SET_ULP_CKSUM      0x00002000      /* Calculate ULP checksum */
2229 #define IXAF_VERIFY_SOURCE      0x00004000      /* Check that source is ok */
2230 #define IXAF_NEXTHOP_SET        0x00008000      /* ixa_nexthop set */

2232 #define IXAF_PMTU_IPV4_DF       0x00010000      /* Set IPv4 DF */
2233 #define IXAF_NO_DEV_FLOW_CTL    0x00020000      /* Protocol needs no flow ctl */
2234 #define IXAF_NO_TTL_CHANGE      0x00040000      /* Internal to IP */
2235 #define IXAF_IPV6_ADD_FRAGHDR   0x00080000      /* Add fragment header */

2237 #define IXAF_IPSEC_TUNNEL       0x00100000      /* Tunnel mode */
2238 #define IXAF_NO_PFHOOK          0x00200000      /* Skip xmit pfhook */
2239 #define IXAF_NO_TRACE           0x00400000      /* When back from ARP/ND */
```

```
#define IXAF_SCOPEID_SET        0x00800000      /* ixa_scopeid set */

#define IXAF_MULTIRT_MULTICAST  0x01000000      /* MULTIRT for multicast */
#define IXAF_NO_HW_CKSUM        0x02000000      /* Force software cksum */
#define IXAF_SET_RAW_CKSUM      0x04000000      /* Use ixa_raw_cksum_offset */
#define IXAF_IPSEC_GLOBAL_POLICY 0x08000000     /* Policy came from global */

/* Note the following uses bits 0x10000000 through 0x80000000 */
#define IXAF_IS_IPV4            IAF_IS_IPV4
#define IXAF_TRUSTED_ICMP       IAF_TRUSTED_ICMP
#define IXAF_NO_LOOP_ZONEID_SET IAF_NO_LOOP_ZONEID_SET
#define IXAF_LOOPBACK_COPY      IAF_LOOPBACK_COPY

/* Note: use the upper 32 bits */
#define IXAF_VERIFY_LSO         0x100000000     /* Check LSO capability */
#define IXAF_LSO_CAPAB          0x200000000     /* Capable of LSO */
#define IXAF_VERIFY_ZCOPY       0x400000000     /* Check Zero Copy capability */
#define IXAF_ZCOPY_CAPAB        0x800000000     /* Capable of ZEROCOPY */

/*
 * The normal flags for sending packets e.g., icmp errors
 */
#define IXAF_BASIC_SIMPLE_V4    \
        (IXAF_SET_ULP_CKSUM | IXAF_IS_IPV4 | IXAF_VERIFY_SOURCE)
#define IXAF_BASIC_SIMPLE_V6    (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE)

/*
 * Normally these fields do not have a hold. But in some cases they do, for
 * instance when we've gone through ip_*_attr_to/from_mblk.
 * We use ixa_free_flags to indicate that they have a hold and need to be
 * released on cleanup.
 */
#define IXA_FREE_CRED           0x00000001      /* ixa_cred needs to be rele */
#define IXA_FREE_TSL            0x00000002      /* ixa_tsl needs to be rele */

/*
 * Simplistic way to set the ixa_xmit_hint for locally generated traffic
 * and forwarded traffic. The shift amount are based on the size of the
 * structs to discard the low order bits which don't have much if any variation
 * (coloring in kmem_cache_alloc might provide some variation).
 *
 * Basing the locally generated hint on the address of the conn_t means that
 * the packets from the same socket/connection do not get reordered.
 * Basing the hint for forwarded traffic on the ill_ring_t means that
 * packets from the same NIC+ring are likely to use the same outbound ring
 * hence we get low contention on the ring in the transmitting driver.
 */
#define CONN_TO_XMIT_HINT(connp)        ((uint32_t)(((uintptr_t)connp) >> 11))
#define ILL_RING_TO_XMIT_HINT(ring)     ((uint32_t)(((uintptr_t)ring) >> 7))

/*
 * IP set Destination Flags used by function ip_set_destination,
 * ip_attr_connect, and conn_connect.
 */
#define IPDF_ALLOW_MCBC         0x1     /* Allow multi/broadcast */
#define IPDF_VERIFY_DST         0x2     /* Verify destination addr */
#define IPDF_SELECT_SRC         0x4     /* Select source address */
#define IPDF_LSO                0x8     /* Try LSO */
#define IPDF_IPSEC              0x10    /* Set IPsec policy */
#define IPDF_ZONE_IS_GLOBAL     0x20    /* From conn_zone_is_global */
#define IPDF_ZCOPY              0x40    /* Try ZEROCOPY */
#define IPDF_UNIQUE_DCE         0x80    /* Get a per-destination DCE */

/*
 * Receive side attributes used between the transport protocols and IP as
 * well as inside IP.
```



```
 */
struct ip_recv_attr_s {
        iaflags_t       ira_flags;      /* See below */

        uint32_t        ira_free_flags; /* IRA_FREE_*. See below */

        /*
         * This is a hint for TCP SYN packets.
         * Always initialized independently of ira_flags settings
         */
        squeue_t        *ira_sqp;
        ill_rx_ring_t   *ira_ring;      /* Internal to IP */

        /* For ip_accept_tcp when IRAF_TARGET_SQP is set */
        squeue_t        *ira_target_sqp;
        mblk_t          *ira_target_sqp_mp;

        /* Always initialized independently of ira_flags settings */
        uint32_t        ira_xmit_hint;  /* For ECMP and GLD TX ring fanout */
        zoneid_t        ira_zoneid;     /* ALL_ZONES unless local delivery */
        uint_t          ira_pktlen;     /* Always set. For frag and stats */
        uint16_t        ira_ip_hdr_length; /* Points to ULP header */
        uint8_t         ira_protocol;   /* Protocol number for ULP cksum */
        uint_t          ira_rifindex;   /* Received ifindex */
        uint_t          ira_ruifindex;  /* Received upper ifindex */
        ts_label_t      *ira_tsl;       /* Always set. NULL if not TX */
        /*
         * ira_rill and ira_ill is set inside IP, but not when conn_recv is
         * called; ULPs should use ira_ruifindex instead.
         */
        ill_t           *ira_rill;      /* ill where packet came */
        ill_t           *ira_ill;       /* ill where IP address hosted */
        cred_t          *ira_cred;      /* For getpeerucred */
        pid_t           ira_cpid;       /* For getpeerucred */

        /* Used when IRAF_VERIFIED_SRC is set; this source was ok */
        ipaddr_t        ira_verified_src;

        /*
         * The following IPsec fields are only initialized when
         * IRAF_IPSEC_SECURE is set. Otherwise they contain garbage.
         */
        struct ipsec_action_s *ira_ipsec_action; /* how we made it in.. */
        struct ipsa_s   *ira_ipsec_ah_sa;       /* SA for AH */
        struct ipsa_s   *ira_ipsec_esp_sa;      /* SA for ESP */

        ipaddr_t        ira_mroute_tunnel;      /* IRAF_MROUTE_TUNNEL_SET */

        zoneid_t        ira_no_loop_zoneid;     /* IRAF_NO_LOOP_ZONEID_SET */

        uint32_t        ira_esp_udp_ports;      /* IRAF_ESP_UDP_PORTS */

        /*
         * For IP_RECVSLLA and ip_ndp_conflict/find_solicitation.
         * Same size as max for sockaddr_dl
         */
#define IRA_L2SRC_SIZE  244
        uint8_t         ira_l2src[IRA_L2SRC_SIZE];      /* If IRAF_L2SRC_SET */

        /*
         * Local handle that we use to do lazy setting of ira_l2src.
         * We defer setting l2src until needed but we do before any
         * ip_input pullupmsg or copymsg.
         */
        struct mac_header_info_s *ira_mhip;     /* Could be NULL */
};
```

```
/*
 * Flags to indicate which receive attributes are set.
 */
#define IRAF_SYSTEM_LABELED     0x00000001      /* is_system_labeled() */
#define IRAF_IPV4_OPTIONS       0x00000002      /* Performance */
#define IRAF_MULTICAST          0x00000004      /* Was multicast at L3 */
#define IRAF_BROADCAST          0x00000008      /* Was broadcast at L3 */
#define IRAF_MULTIBROADCAST     (IRAF_MULTICAST|IRAF_BROADCAST)

#define IRAF_LOOPBACK           0x00000010      /* Looped back by IP */
#define IRAF_VERIFY_IP_CKSUM    0x00000020      /* Need to verify IP */
#define IRAF_VERIFY_ULP_CKSUM   0x00000040      /* Need to verify TCP,UDP,etc */
#define IRAF_SCTP_CSUM_ERR      0x00000080      /* sctp pkt has failed chksum */

#define IRAF_IPSEC_SECURE       0x00000100      /* Passed AH and/or ESP */
#define IRAF_DHCP_UNICAST       0x00000200
#define IRAF_IPSEC_DECAPS       0x00000400      /* Was packet decapsulated */
                                        /* from a matching inner packet? */
#define IRAF_TARGET_SQP         0x00000800      /* ira_target_sqp is set */
#define IRAF_VERIFIED_SRC       0x00001000      /* ira_verified_src set */
#define IRAF_RSVP               0x00002000      /* RSVP packet for rsvpd */
#define IRAF_MROUTE_TUNNEL_SET  0x00004000      /* From ip_mroute_decap */
#define IRAF_PIM_REGISTER       0x00008000      /* From register_mforward */

#define IRAF_TX_MAC_EXEMPTABLE  0x00010000      /* Allow MAC_EXEMPT readdown */
#define IRAF_TX_SHARED_ADDR     0x00020000      /* Arrived on ALL_ZONES addr */
#define IRAF_ESP_UDP_PORTS      0x00040000      /* NAT-traversal packet */
#define IRAF_NO_HW_CKSUM        0x00080000      /* Force software cksum */

#define IRAF_ICMP_ERROR         0x00100000      /* Send to conn_recvicmp */
#define IRAF_ROUTER_ALERT       0x00200000      /* IPv6 router alert */
#define IRAF_L2SRC_SET          0x00400000      /* ira_l2src has been set */
#define IRAF_L2SRC_LOOPBACK     0x00800000      /* Came from us */

#define IRAF_L2DST_MULTICAST    0x01000000      /* Multicast at L2 */
#define IRAF_L2DST_BROADCAST    0x02000000      /* Broadcast at L2 */
/* Unused 0x04000000 */
/* Unused 0x08000000 */

/* Below starts with 0x10000000 */
#define IRAF_IS_IPV4            IAF_IS_IPV4
#define IRAF_TRUSTED_ICMP       IAF_TRUSTED_ICMP
#define IRAF_NO_LOOP_ZONEID_SET IAF_NO_LOOP_ZONEID_SET
#define IRAF_LOOPBACK_COPY      IAF_LOOPBACK_COPY

/*
 * Normally these fields do not have a hold. But in some cases they do, for
 * instance when we've gone through ip_*_attr_to/from_mblk.
 * We use ira_free_flags to indicate that they have a hold and need to be
 * released on cleanup.
 */
#define IRA_FREE_CRED           0x00000001      /* ira_cred needs to be rele */
#define IRA_FREE_TSL            0x00000002      /* ira_tsl needs to be rele */

/*
 * Optional destination cache entry for path MTU information,
 * and ULP metrics.
 */
struct dce_s {
        uint_t          dce_generation; /* Changed since cached? */
        uint_t          dce_flags;      /* See below */
        uint_t          dce_ipversion;  /* IPv4/IPv6 version */
        uint32_t        dce_pmtu;       /* Path MTU if DCEF_PMTU */
        uint32_t        dce_ident;      /* Per destination IP ident. */
        iulp_t          dce_uinfo;      /* Metrics if DCEF_UINFO */
```



```
        struct dce_s    *dce_next;
        struct dce_s    **dce_ptpn;
        struct dcb_s    *dce_bucket;

        union {
                in6_addr_t      dceu_v6addr;
                ipaddr_t        dceu_v4addr;
        } dce_u;
#define dce_v4addr      dce_u.dceu_v4addr
#define dce_v6addr      dce_u.dceu_v6addr
        /* Note that for IPv6+IPMP we use the ifindex for the upper interface */
        uint_t          dce_ifindex;    /* For IPv6 link-locals */

        kmutex_t        dce_lock;
        uint_t          dce_refcnt;
        uint64_t        dce_last_change_time;   /* Path MTU. In seconds */

        ip_stack_t      *dce_ipst;      /* Does not have a netstack_hold */
};

/*
 * Values for dce_generation.
 *
 * If a DCE has DCE_GENERATION_CONDEMNED, the last dce_refrele should delete
 * it.
 *
 * DCE_GENERATION_VERIFY is never stored in dce_generation but it is
 * stored in places that cache DCE (such as ixa_dce_generation).
 * It is used as a signal that the cache is stale and needs to be reverified.
 */
#define DCE_GENERATION_CONDEMNED        0
#define DCE_GENERATION_VERIFY           1
#define DCE_GENERATION_INITIAL          2
#define DCE_IS_CONDEMNED(dce) \
        ((dce)->dce_generation == DCE_GENERATION_CONDEMNED)


/*
 * Values for ips_src_generation.
 *
 * SRC_GENERATION_VERIFY is never stored in ips_src_generation but it is
 * stored in places that cache IREs (ixa_src_generation). It is used as a
 * signal that the cache is stale and needs to be reverified.
 */
#define SRC_GENERATION_VERIFY           0
#define SRC_GENERATION_INITIAL          1

/*
 * The kernel stores security attributes of all gateways in a database made
 * up of one or more tsol_gcdb_t elements.  Each tsol_gcdb_t contains the
 * security-related credentials of the gateway.  More than one gateways may
 * share entries in the database.
 *
 * The tsol_gc_t structure represents the gateway to credential association,
 * and refers to an entry in the database.  One or more tsol_gc_t entities are
 * grouped together to form one or more tsol_gcgrp_t, each representing the
 * list of security attributes specific to the gateway.  A gateway may be
 * associated with at most one credentials group.
 */
struct tsol_gcgrp_s;

extern uchar_t  ip6opt_ls;      /* TX IPv6 enabler */

/*
 * Gateway security credential record.
```

```
 */
typedef struct tsol_gcdb_s {
        uint_t          gcdb_refcnt;    /* reference count */
        struct rtsa_s   gcdb_attr;      /* security attributes */
#define gcdb_mask       gcdb_attr.rtsa_mask
#define gcdb_doi        gcdb_attr.rtsa_doi
#define gcdb_slrange    gcdb_attr.rtsa_slrange
} tsol_gcdb_t;

/*
 * Gateway to credential association.
 */
typedef struct tsol_gc_s {
        uint_t          gc_refcnt;      /* reference count */
        struct tsol_gcgrp_s *gc_grp;    /* pointer to group */
        struct tsol_gc_s *gc_prev;      /* previous in list */
        struct tsol_gc_s *gc_next;      /* next in list */
        tsol_gcdb_t     *gc_db;         /* pointer to actual credentials */
} tsol_gc_t;

/*
 * Gateway credentials group address.
 */
typedef struct tsol_gcgrp_addr_s {
        int             ga_af;          /* address family */
        in6_addr_t      ga_addr;        /* IPv4 mapped or IPv6 address */
} tsol_gcgrp_addr_t;

/*
 * Gateway credentials group.
 */
typedef struct tsol_gcgrp_s {
        uint_t          gcgrp_refcnt;   /* reference count */
        krwlock_t       gcgrp_rwlock;   /* lock to protect following */
        uint_t          gcgrp_count;    /* number of credentials */
        tsol_gc_t       *gcgrp_head;    /* first credential in list */
        tsol_gc_t       *gcgrp_tail;    /* last credential in list */
        tsol_gcgrp_addr_t gcgrp_addr;   /* next-hop gateway address */
} tsol_gcgrp_t;

extern kmutex_t gcgrp_lock;

#define GC_REFRELE(p) {                         \
        ASSERT((p)->gc_grp != NULL);            \
        rw_enter(&(p)->gc_grp->gcgrp_rwlock, RW_WRITER); \
        ASSERT((p)->gc_refcnt > 0);             \
        if (--((p)->gc_refcnt) == 0)            \
                gc_inactive(p);                 \
        else                                    \
                rw_exit(&(p)->gc_grp->gcgrp_rwlock); \
}

#define GCGRP_REFHOLD(p) {                      \
        mutex_enter(&gcgrp_lock);               \
        ++((p)->gcgrp_refcnt);                  \
        ASSERT((p)->gcgrp_refcnt != 0);         \
        mutex_exit(&gcgrp_lock);                \
}

#define GCGRP_REFRELE(p) {                      \
        mutex_enter(&gcgrp_lock);               \
        ASSERT((p)->gcgrp_refcnt > 0);          \
        if (--((p)->gcgrp_refcnt) == 0)         \
                gcgrp_inactive(p);              \
        ASSERT(MUTEX_HELD(&gcgrp_lock));        \
        mutex_exit(&gcgrp_lock);                \
```



```
}

/*
 * IRE gateway security attributes structure, pointed to by tsol_ire_gw_secattr
 */
struct tsol_tnrhc;

struct tsol_ire_gw_secattr_s {
        kmutex_t        igsa_lock;      /* lock to protect following */
        struct tsol_tnrhc *igsa_rhc;    /* host entry for gateway */
        tsol_gc_t       *igsa_gc;       /* for prefix IREs */
};

void irb_refrele_ftable(irb_t *);

extern struct kmem_cache *rt_entry_cache;

typedef struct ire4 {
        ipaddr_t ire4_mask;             /* Mask for matching this IRE. */
        ipaddr_t ire4_addr;             /* Address this IRE represents. */
        ipaddr_t ire4_gateway_addr;     /* Gateway including for IRE_ONLINK */
        ipaddr_t ire4_setsrc_addr;      /* RTF_SETSRC */
} ire4_t;

typedef struct ire6 {
        in6_addr_t ire6_mask;           /* Mask for matching this IRE. */
        in6_addr_t ire6_addr;           /* Address this IRE represents. */
        in6_addr_t ire6_gateway_addr;   /* Gateway including for IRE_ONLINK */
        in6_addr_t ire6_setsrc_addr;    /* RTF_SETSRC */
} ire6_t;

typedef union ire_addr {
        ire6_t  ire6_u;
        ire4_t  ire4_u;
} ire_addr_u_t;

/*
 * Internet Routing Entry
 * When we have multiple identical IREs we logically add them by manipulating
 * ire_identical_ref and ire_delete first decrements
 * that and when it reaches 1 we know it is the last IRE.
 * "identical" is defined as being the same for:
 * ire_addr, ire_netmask, ire_gateway, ire_ill, ire_zoneid, and ire_type
 * For instance, multiple IRE_BROADCASTs for the same subnet number are
 * viewed as identical, and so are the IRE_INTERFACEs when there are
 * multiple logical interfaces (on the same ill) with the same subnet prefix.
 */
struct ire_s {
        struct  ire_s   *ire_next;      /* The hash chain must be first. */
        struct  ire_s   **ire_ptpn;     /* Pointer to previous next. */
        uint32_t        ire_refcnt;     /* Number of references */
        ill_t           *ire_ill;
        uint32_t        ire_identical_ref; /* IRE_INTERFACE, IRE_BROADCAST */
        uchar_t         ire_ipversion;  /* IPv4/IPv6 version */
        ushort_t        ire_type;       /* Type of IRE */
        uint_t          ire_generation; /* Generation including CONDEMNED */
        uint_t  ire_ib_pkt_count;       /* Inbound packets for ire_addr */
        uint_t  ire_ob_pkt_count;       /* Outbound packets to ire_addr */
        time_t  ire_create_time;        /* Time (in secs) IRE was created. */
        uint32_t        ire_flags;      /* flags related to route (RTF_*) */
        /*
         * ire_testhidden is TRUE for INTERFACE IREs of IS_UNDER_IPMP(ill)
         * interfaces
         */
        boolean_t       ire_testhidden;
        pfirerecv_t     ire_recvfn;     /* Receive side handling */
```

```
2636         pfiresend_t     ire_sendfn;     /* Send side handling */
2637         pfirepostfrag_t ire_postfragfn; /* Bottom end of send handling */

2639         uint_t          ire_masklen;    /* # bits in ire_mask{,_v6} */
2640         ire_addr_u_t    ire_u;          /* IPv4/IPv6 address info. */

2642         irb_t           *ire_bucket;    /* Hash bucket when ire_ptphn is set */
2643         kmutex_t        ire_lock;
2644         clock_t         ire_last_used_time;     /* For IRE_LOCAL reception */
2645         tsol_ire_gw_secattr_t *ire_gw_secattr; /* gateway security attributes */
2646         zoneid_t        ire_zoneid;

2648         /*
2649          * Cached information of where to send packets that match this route.
2650          * The ire_dep_* information is used to determine when ire_nce_cache
2651          * needs to be updated.
2652          * ire_nce_cache is the fastpath for the Neighbor Cache Entry
2653          * for IPv6; arp info for IPv4
2654          * Since this is a cache setup and torn down independently of
2655          * applications we need to use nce_ref{rele,hold}_notr for it.
2656          */
2657         nce_t           *ire_nce_cache;

2659         /*
2660          * Quick check whether the ire_type and ire_masklen indicates
2661          * that the IRE can have ire_nce_cache set i.e., whether it is
2662          * IRE_ONLINK and for a single destination.
2663          */
2664         boolean_t       ire_nce_capable;

2666         /*
2667          * Dependency tracking so we can safely cache IRE and NCE pointers
2668          * in offlink and onlink IREs.
2669          * These are locked under the ips_ire_dep_lock rwlock. Write held
2670          * when modifying the linkage.
2671          * ire_dep_parent (Also chain towards IRE for nexthop)
2672          * ire_dep_parent_generation: ire_generation of ire_dep_parent
2673          * ire_dep_children (From parent to first child)
2674          * ire_dep_sib_next (linked list of siblings)
2675          * ire_dep_sib_ptpn (linked list of siblings)
2676          *
2677          * The parent has a ire_refhold on each child, and each child has
2678          * an ire_refhold on its parent.
2679          * Since ire_dep_parent is a cache setup and torn down independently of
2680          * applications we need to use ire_ref{rele,hold}_notr for it.
2681          */
2682         ire_t           *ire_dep_parent;
2683         ire_t           *ire_dep_children;
2684         ire_t           *ire_dep_sib_next;
2685         ire_t           **ire_dep_sib_ptpn;     /* Pointer to previous next */
2686         uint_t          ire_dep_parent_generation;

2688         uint_t          ire_badcnt;     /* Number of times ND_UNREACHABLE */
2689         uint64_t        ire_last_badcnt;        /* In seconds */

2691         /* ire_defense* and ire_last_used_time are only used on IRE_LOCALs */
2692         uint_t          ire_defense_count;      /* number of ARP conflicts */
2693         uint_t          ire_defense_time;       /* last time defended (secs) */

2695         boolean_t       ire_trace_disable;      /* True when alloc fails */
2696         ip_stack_t      *ire_ipst;      /* Does not have a netstack_hold */
2697         iulp_t          ire_metrics;
2698         /*
2699          * default and prefix routes that are added without explicitly
2700          * specifying the interface are termed "unbound" routes, and will
2701          * have ire_unbound set to true.
```



```
2702          */
2703         boolean_t       ire_unbound;
2704 };

2706 /* IPv4 compatibility macros */
2707 #define ire_mask                ire_u.ire4_u.ire4_mask
2708 #define ire_addr                ire_u.ire4_u.ire4_addr
2709 #define ire_gateway_addr        ire_u.ire4_u.ire4_gateway_addr
2710 #define ire_setsrc_addr         ire_u.ire4_u.ire4_setsrc_addr

2712 #define ire_mask_v6             ire_u.ire6_u.ire6_mask
2713 #define ire_addr_v6             ire_u.ire6_u.ire6_addr
2714 #define ire_gateway_addr_v6     ire_u.ire6_u.ire6_gateway_addr
2715 #define ire_setsrc_addr_v6      ire_u.ire6_u.ire6_setsrc_addr

2717 /*
2718  * Values for ire_generation.
2719  *
2720  * If an IRE is marked with IRE_IS_CONDEMNED, the last walker of
2721  * the bucket should delete this IRE from this bucket.
2722  *
2723  * IRE_GENERATION_VERIFY is never stored in ire_generation but it is
2724  * stored in places that cache IREs (such as ixa_ire_generation and
2725  * ire_dep_parent_generation). It is used as a signal that the cache is
2726  * stale and needs to be reverified.
2727  */
2728 #define IRE_GENERATION_CONDEMNED        0
2729 #define IRE_GENERATION_VERIFY           1
2730 #define IRE_GENERATION_INITIAL          2
2731 #define IRE_IS_CONDEMNED(ire) \
2732         ((ire)->ire_generation == IRE_GENERATION_CONDEMNED)

2734 /* Convenient typedefs for sockaddrs */
2735 typedef struct sockaddr_in      sin_t;
2736 typedef struct sockaddr_in6     sin6_t;

2738 /* Name/Value Descriptor. */
2739 typedef struct nv_s {
2740         uint64_t nv_value;
2741         char    *nv_name;
2742 } nv_t;

2744 #define ILL_FRAG_HASH(s, i) \
2745         ((ntohl(s) ^ ((i) ^ ((i) >> 8))) % ILL_FRAG_HASH_TBL_COUNT)

2747 /*
2748  * The MAX number of allowed fragmented packets per hash bucket
2749  * calculation is based on the most common mtu size of 1500. This limit
2750  * will work well for other mtu sizes as well.
2751  */
2752 #define COMMON_IP_MTU 1500
2753 #define MAX_FRAG_MIN 10
2754 #define MAX_FRAG_PKTS(ipst)     \
2755         MAX(MAX_FRAG_MIN, (2 * (ipst->ips_ip_reass_queue_bytes / \
2756             (COMMON_IP_MTU * ILL_FRAG_HASH_TBL_COUNT))))

2758 /*
2759  * Maximum dups allowed per packet.
2760  */
2761 extern uint_t ip_max_frag_dups;

2763 /*
2764  * Per-packet information for received packets and transmitted.
2765  * Used by the transport protocols when converting between the packet
2766  * and ancillary data and socket options.
2767  *
```

```
 * Note: This private data structure and related IPPF_* constant
 * definitions are exposed to enable compilation of some debugging tools
 * like lsof which use struct tcp_t in <inet/tcp.h>. This is intended to be
 * a temporary hack and long term alternate interfaces should be defined
 * to support the needs of such tools and private definitions moved to
 * private headers.
 */
struct ip_pkt_s {
        uint_t          ipp_fields;             /* Which fields are valid */
        in6_addr_t      ipp_addr;               /* pktinfo src/dst addr */
#define ipp_addr_v4     V4_PART_OF_V6(ipp_addr)
        uint_t          ipp_unicast_hops;       /* IPV6_UNICAST_HOPS, IP_TTL */
        uint_t          ipp_hoplimit;           /* IPV6_HOPLIMIT */
        uint_t          ipp_hopoptslen;
        uint_t          ipp_rthdrdstoptslen;
        uint_t          ipp_rthdrlen;
        uint_t          ipp_dstoptslen;
        uint_t          ipp_fraghdrlen;
        ip6_hbh_t       *ipp_hopopts;
        ip6_dest_t      *ipp_rthdrdstopts;
        ip6_rthdr_t     *ipp_rthdr;
        ip6_dest_t      *ipp_dstopts;
        ip6_frag_t      *ipp_fraghdr;
        uint8_t         ipp_tclass;             /* IPV6_TCLASS */
        uint8_t         ipp_type_of_service;    /* IP_TOS */
        uint_t          ipp_ipv4_options_len;   /* Len of IPv4 options */
        uint8_t         *ipp_ipv4_options;      /* Ptr to IPv4 options */
        uint_t          ipp_label_len_v4;       /* Len of TX label for IPv4 */
        uint8_t         *ipp_label_v4;          /* TX label for IPv4 */
        uint_t          ipp_label_len_v6;       /* Len of TX label for IPv6 */
        uint8_t         *ipp_label_v6;          /* TX label for IPv6 */
};
typedef struct ip_pkt_s ip_pkt_t;

extern void ip_pkt_free(ip_pkt_t *);    /* free storage inside ip_pkt_t */
extern ipaddr_t ip_pkt_source_route_v4(const ip_pkt_t *);
extern in6_addr_t *ip_pkt_source_route_v6(const ip_pkt_t *);
extern int ip_pkt_copy(ip_pkt_t *, ip_pkt_t *, int);
extern void ip_pkt_source_route_reverse_v4(ip_pkt_t *);

/* ipp_fields values */
#define IPPF_ADDR               0x0001  /* Part of in6_pktinfo: src/dst addr */
#define IPPF_HOPLIMIT           0x0002  /* Overrides unicast and multicast */
#define IPPF_TCLASS             0x0004  /* Overrides class in sin6_flowinfo */

#define IPPF_HOPOPTS            0x0010  /* ipp_hopopts set */
#define IPPF_RTHDR              0x0020  /* ipp_rthdr set */
#define IPPF_RTHDRDSTOPTS       0x0040  /* ipp_rthdrdstopts set */
#define IPPF_DSTOPTS            0x0080  /* ipp_dstopts set */

#define IPPF_IPV4_OPTIONS       0x0100  /* ipp_ipv4_options set */
#define IPPF_LABEL_V4           0x0200  /* ipp_label_v4 set */
#define IPPF_LABEL_V6           0x0400  /* ipp_label_v6 set */

#define IPPF_FRAGHDR            0x0800  /* Used for IPsec receive side */

/*
 * Data structure which is passed to conn_opt_get/set.
 * The conn_t is included even though it can be inferred from queue_t.
 * setsockopt and getsockopt use conn_ixa and conn_xmit_ipp. However,
 * when handling ancillary data we use separate ixa and ipps.
 */
typedef struct conn_opt_arg_s {
        conn_t          *coa_connp;
        ip_xmit_attr_t  *coa_ixa;
        ip_pkt_t        *coa_ipp;
```



```
        boolean_t       coa_ancillary;  /* Ancillary data and not setsockopt */
        uint_t          coa_changed;    /* See below */
} conn_opt_arg_t;

/*
 * Flags for what changed.
 * If we want to be more efficient in the future we can have more fine
 * grained flags e.g., a flag for just IP_TOS changing.
 * For now we either call ip_set_destination (for "route changed")
 * and/or conn_build_hdr_template/conn_prepend_hdr (for "header changed").
 */
#define COA_HEADER_CHANGED      0x0001
#define COA_ROUTE_CHANGED       0x0002
#define COA_RCVBUF_CHANGED      0x0004  /* SO_RCVBUF */
#define COA_SNDBUF_CHANGED      0x0008  /* SO_SNDBUF */
#define COA_WROFF_CHANGED       0x0010  /* Header size changed */
#define COA_ICMP_BIND_NEEDED    0x0020
#define COA_OOBINLINE_CHANGED   0x0040

#define TCP_PORTS_OFFSET        0
#define UDP_PORTS_OFFSET        0

/*
 * lookups return the ill/ipif only if the flags are clear OR Iam writer.
 * ill / ipif lookup functions increment the refcnt on the ill / ipif only
 * after calling these macros. This ensures that the refcnt on the ipif or
 * ill will eventually drop down to zero.
 */
#define ILL_LOOKUP_FAILED       1       /* Used as error code */
#define IPIF_LOOKUP_FAILED      2       /* Used as error code */

#define ILL_CAN_LOOKUP(ill)                                             \
        (!((ill)->ill_state_flags & ILL_CONDEMNED) ||                   \
        IAM_WRITER_ILL(ill))

#define ILL_IS_CONDEMNED(ill)   \
        ((ill)->ill_state_flags & ILL_CONDEMNED)

#define IPIF_CAN_LOOKUP(ipif)   \
        (!((ipif)->ipif_state_flags & IPIF_CONDEMNED) || \
        IAM_WRITER_IPIF(ipif))

#define IPIF_IS_CONDEMNED(ipif) \
        ((ipif)->ipif_state_flags & IPIF_CONDEMNED)

#define IPIF_IS_CHANGING(ipif)  \
        ((ipif)->ipif_state_flags & IPIF_CHANGING)

/* Macros used to assert that this thread is a writer */
#define IAM_WRITER_IPSQ(ipsq)   ((ipsq)->ipsq_xop->ipx_writer == curthread)
#define IAM_WRITER_ILL(ill)     IAM_WRITER_IPSQ((ill)->ill_phyint->phyint_ipsq)
#define IAM_WRITER_IPIF(ipif)   IAM_WRITER_ILL((ipif)->ipif_ill)

/*
 * Grab ill locks in the proper order. The order is highest addressed
 * ill is locked first.
 */
#define GRAB_ILL_LOCKS(ill_1, ill_2)                                    \
{                                                                       \
        if ((ill_1) > (ill_2)) {                                        \
                if (ill_1 != NULL)                                      \
                        mutex_enter(&(ill_1)->ill_lock);                \
                if (ill_2 != NULL)                                      \
                        mutex_enter(&(ill_2)->ill_lock);                \
        } else {                                                        \
                if (ill_2 != NULL)                                      \
```

```
2900                        mutex_enter(&(ill_2)->ill_lock);        \
2901                if (ill_1 != NULL && ill_1 != ill_2)            \
2902                        mutex_enter(&(ill_1)->ill_lock);        \
2903        }                                                       \
2904 }

2906 #define RELEASE_ILL_LOCKS(ill_1, ill_2)         \
2907 {                                               \
2908        if (ill_1 != NULL)                       \
2909                mutex_exit(&(ill_1)->ill_lock); \
2910        if (ill_2 != NULL && ill_2 != ill_1)     \
2911                mutex_exit(&(ill_2)->ill_lock); \
2912 }

2914 /* Get the other protocol instance ill */
2915 #define ILL_OTHER(ill)                                  \
2916        ((ill)->ill_isv6 ? (ill)->ill_phyint->phyint_illv4 :    \
2917            (ill)->ill_phyint->phyint_illv6)

2919 /* ioctl command info: Ioctl properties extracted and stored in here */
2920 typedef struct cmd_info_s
2921 {
2922        ipif_t  *ci_ipif;       /* ipif associated with [l]ifreq ioctl's */
2923        sin_t   *ci_sin;        /* the sin struct passed down */
2924        sin6_t  *ci_sin6;       /* the sin6_t struct passed down */
2925        struct lifreq *ci_lifr; /* the lifreq struct passed down */
2926 } cmd_info_t;

2928 extern struct kmem_cache *ire_cache;

2930 extern ipaddr_t ip_g_all_ones;

2932 extern uint_t   ip_loopback_mtu;        /* /etc/system */
2933 extern uint_t   ip_loopback_mtuplus;
2934 extern uint_t   ip_loopback_mtu_v6plus;

2936 extern vmem_t *ip_minor_arena_sa;
2937 extern vmem_t *ip_minor_arena_la;

2939 /*
2940  * ip_g_forward controls IP forwarding.  It takes two values:
2941  *      0: IP_FORWARD_NEVER     Don't forward packets ever.
2942  *      1: IP_FORWARD_ALWAYS    Forward packets for elsewhere.
2943  *
2944  * RFC1122 says there must be a configuration switch to control forwarding,
2945  * but that the default MUST be to not forward packets ever.  Implicit
2946  * control based on configuration of multiple interfaces MUST NOT be
2947  * implemented (Section 3.1).  SunOS 4.1 did provide the "automatic" capability
2948  * and, in fact, it was the default.  That capability is now provided in the
2949  * /etc/rc2.d/S69inet script.
2950  */

2952 #define ips_ip_respond_to_address_mask_broadcast \
2953                                         ips_propinfo_tbl[0].prop_cur_bval
2954 #define ips_ip_g_resp_to_echo_bcast     ips_propinfo_tbl[1].prop_cur_bval
2955 #define ips_ip_g_resp_to_echo_mcast     ips_propinfo_tbl[2].prop_cur_bval
2956 #define ips_ip_g_resp_to_timestamp      ips_propinfo_tbl[3].prop_cur_bval
2957 #define ips_ip_g_resp_to_timestamp_bcast ips_propinfo_tbl[4].prop_cur_bval
2958 #define ips_ip_g_send_redirects         ips_propinfo_tbl[5].prop_cur_bval
2959 #define ips_ip_g_forward_directed_bcast ips_propinfo_tbl[6].prop_cur_bval
2960 #define ips_ip_mrtdebug                 ips_propinfo_tbl[7].prop_cur_uval
2961 #define ips_ip_ire_reclaim_fraction     ips_propinfo_tbl[8].prop_cur_uval
2962 #define ips_ip_nce_reclaim_fraction     ips_propinfo_tbl[9].prop_cur_uval
2963 #define ips_ip_dce_reclaim_fraction     ips_propinfo_tbl[10].prop_cur_uval
2964 #define ips_ip_def_ttl                  ips_propinfo_tbl[11].prop_cur_uval
2965 #define ips_ip_forward_src_routed       ips_propinfo_tbl[12].prop_cur_bval
```



```
2966 #define ips_ip_wroff_extra              ips_propinfo_tbl[13].prop_cur_uval
2967 #define ips_ip_pathmtu_interval         ips_propinfo_tbl[14].prop_cur_uval
2968 #define ips_ip_icmp_return              ips_propinfo_tbl[15].prop_cur_uval
2969 #define ips_ip_path_mtu_discovery       ips_propinfo_tbl[16].prop_cur_bval
2970 #define ips_ip_pmtu_min                 ips_propinfo_tbl[17].prop_cur_uval
2971 #define ips_ip_ignore_redirect          ips_propinfo_tbl[18].prop_cur_bval
2972 #define ips_ip_arp_icmp_error           ips_propinfo_tbl[19].prop_cur_bval
2973 #define ips_ip_broadcast_ttl            ips_propinfo_tbl[20].prop_cur_uval
2974 #define ips_ip_icmp_err_interval        ips_propinfo_tbl[21].prop_cur_uval
2975 #define ips_ip_icmp_err_burst           ips_propinfo_tbl[22].prop_cur_uval
2976 #define ips_ip_reass_queue_bytes        ips_propinfo_tbl[23].prop_cur_uval
2977 #define ips_ip_strict_dst_multihoming   ips_propinfo_tbl[24].prop_cur_uval
2978 #define ips_ip_addrs_per_if             ips_propinfo_tbl[25].prop_cur_uval
2979 #define ips_ipsec_override_persocket_policy ips_propinfo_tbl[26].prop_cur_bval
2980 #define ips_icmp_accept_clear_messages  ips_propinfo_tbl[27].prop_cur_bval
2981 #define ips_igmp_accept_clear_messages  ips_propinfo_tbl[28].prop_cur_bval

2983 /* IPv6 configuration knobs */
2984 #define ips_delay_first_probe_time      ips_propinfo_tbl[29].prop_cur_uval
2985 #define ips_max_unicast_solicit         ips_propinfo_tbl[30].prop_cur_uval
2986 #define ips_ipv6_def_hops               ips_propinfo_tbl[31].prop_cur_uval
2987 #define ips_ipv6_icmp_return            ips_propinfo_tbl[32].prop_cur_uval
2988 #define ips_ipv6_forward_src_routed     ips_propinfo_tbl[33].prop_cur_bval
2989 #define ips_ipv6_resp_echo_mcast        ips_propinfo_tbl[34].prop_cur_bval
2990 #define ips_ipv6_send_redirects         ips_propinfo_tbl[35].prop_cur_bval
2991 #define ips_ipv6_ignore_redirect        ips_propinfo_tbl[36].prop_cur_bval
2992 #define ips_ipv6_strict_dst_multihoming ips_propinfo_tbl[37].prop_cur_uval
2993 #define ips_src_check                   ips_propinfo_tbl[38].prop_cur_uval
2994 #define ips_ipsec_policy_log_interval   ips_propinfo_tbl[39].prop_cur_uval
2995 #define ips_pim_accept_clear_messages   ips_propinfo_tbl[40].prop_cur_bval
2996 #define ips_ip_ndp_unsolicit_interval   ips_propinfo_tbl[41].prop_cur_uval
2997 #define ips_ip_ndp_unsolicit_count      ips_propinfo_tbl[42].prop_cur_uval
2998 #define ips_ipv6_ignore_home_address_opt ips_propinfo_tbl[43].prop_cur_bval

3000 /* Misc IP configuration knobs */
3001 #define ips_ip_policy_mask              ips_propinfo_tbl[44].prop_cur_uval
3002 #define ips_ip_ecmp_behavior            ips_propinfo_tbl[45].prop_cur_uval
3003 #define ips_ip_multirt_ttl              ips_propinfo_tbl[46].prop_cur_uval
3004 #define ips_ip_ire_badcnt_lifetime      ips_propinfo_tbl[47].prop_cur_uval
3005 #define ips_ip_max_temp_idle            ips_propinfo_tbl[48].prop_cur_uval
3006 #define ips_ip_max_temp_defend          ips_propinfo_tbl[49].prop_cur_uval
3007 #define ips_ip_max_defend               ips_propinfo_tbl[50].prop_cur_uval
3008 #define ips_ip_defend_interval          ips_propinfo_tbl[51].prop_cur_uval
3009 #define ips_ip_dup_recovery             ips_propinfo_tbl[52].prop_cur_uval
3010 #define ips_ip_restrict_interzone_loopback ips_propinfo_tbl[53].prop_cur_bval
3011 #define ips_ip_lso_outbound             ips_propinfo_tbl[54].prop_cur_bval
3012 #define ips_igmp_max_version            ips_propinfo_tbl[55].prop_cur_uval
3013 #define ips_mld_max_version             ips_propinfo_tbl[56].prop_cur_uval
3014 #define ips_ip_forwarding               ips_propinfo_tbl[57].prop_cur_bval
3015 #define ips_ipv6_forwarding             ips_propinfo_tbl[58].prop_cur_bval
3016 #define ips_ip_reassembly_timeout       ips_propinfo_tbl[59].prop_cur_uval
3017 #define ips_ipv6_reassembly_timeout     ips_propinfo_tbl[60].prop_cur_uval
3018 #define ips_ip_cgtp_filter              ips_propinfo_tbl[61].prop_cur_bval
3019 #define ips_arp_probe_delay             ips_propinfo_tbl[62].prop_cur_uval
3020 #define ips_arp_fastprobe_delay         ips_propinfo_tbl[63].prop_cur_uval
3021 #define ips_arp_probe_interval          ips_propinfo_tbl[64].prop_cur_uval
3022 #define ips_arp_fastprobe_interval      ips_propinfo_tbl[65].prop_cur_uval
3023 #define ips_arp_probe_count             ips_propinfo_tbl[66].prop_cur_uval
3024 #define ips_arp_fastprobe_count         ips_propinfo_tbl[67].prop_cur_uval
3025 #define ips_ipv4_dad_announce_interval  ips_propinfo_tbl[68].prop_cur_uval
3026 #define ips_ipv6_dad_announce_interval  ips_propinfo_tbl[69].prop_cur_uval
3027 #define ips_arp_defend_interval         ips_propinfo_tbl[70].prop_cur_uval
3028 #define ips_arp_defend_rate             ips_propinfo_tbl[71].prop_cur_uval
3029 #define ips_ndp_defend_interval         ips_propinfo_tbl[72].prop_cur_uval
3030 #define ips_ndp_defend_rate             ips_propinfo_tbl[73].prop_cur_uval
3031 #define ips_arp_defend_period           ips_propinfo_tbl[74].prop_cur_uval
```

```
3032 #define ips_ndp_defend_period           ips_propinfo_tbl[75].prop_cur_uval
3033 #define ips_ipv4_icmp_return_pmtu       ips_propinfo_tbl[76].prop_cur_bval
3034 #define ips_ipv6_icmp_return_pmtu       ips_propinfo_tbl[77].prop_cur_bval
3035 #define ips_ip_arp_publish_count        ips_propinfo_tbl[78].prop_cur_uval
3036 #define ips_ip_arp_publish_interval     ips_propinfo_tbl[79].prop_cur_uval
3037 #define ips_ip_strict_src_multihoming   ips_propinfo_tbl[80].prop_cur_uval
3038 #define ips_ipv6_strict_src_multihoming ips_propinfo_tbl[81].prop_cur_uval
3039 #define ips_ipv6_drop_inbound_icmpv6    ips_propinfo_tbl[82].prop_cur_bval

3041 extern int      dohwcksum;      /* use h/w cksum if supported by the h/w */
3042 #ifdef ZC_TEST
3043 extern int      noswcksum;
3044 #endif

3046 extern char     ipif_loopback_name[];

3048 extern nv_t     *ire_nv_tbl;

3050 extern struct module_info ip_mod_info;

3052 #define HOOKS4_INTERESTED_PHYSICAL_IN(ipst)     \
3053         ((ipst)->ips_ip4_physical_in_event.he_interested)
3054 #define HOOKS6_INTERESTED_PHYSICAL_IN(ipst)     \
3055         ((ipst)->ips_ip6_physical_in_event.he_interested)
3056 #define HOOKS4_INTERESTED_PHYSICAL_OUT(ipst)    \
3057         ((ipst)->ips_ip4_physical_out_event.he_interested)
3058 #define HOOKS6_INTERESTED_PHYSICAL_OUT(ipst)    \
3059         ((ipst)->ips_ip6_physical_out_event.he_interested)
3060 #define HOOKS4_INTERESTED_FORWARDING(ipst)      \
3061         ((ipst)->ips_ip4_forwarding_event.he_interested)
3062 #define HOOKS6_INTERESTED_FORWARDING(ipst)      \
3063         ((ipst)->ips_ip6_forwarding_event.he_interested)
3064 #define HOOKS4_INTERESTED_LOOPBACK_IN(ipst)     \
3065         ((ipst)->ips_ip4_loopback_in_event.he_interested)
3066 #define HOOKS6_INTERESTED_LOOPBACK_IN(ipst)     \
3067         ((ipst)->ips_ip6_loopback_in_event.he_interested)
3068 #define HOOKS4_INTERESTED_LOOPBACK_OUT(ipst)    \
3069         ((ipst)->ips_ip4_loopback_out_event.he_interested)
3070 #define HOOKS6_INTERESTED_LOOPBACK_OUT(ipst)    \
3071         ((ipst)->ips_ip6_loopback_out_event.he_interested)
3072 /*
3073  * Hooks marcos used inside of ip
3074  * The callers use the above INTERESTED macros first, hence
3075  * the he_interested check is superflous.
3076  */
3077 #define FW_HOOKS(_hook, _event, _ilp, _olp, _iph, _fm, _m, _llm, ipst, _err) \
3078         if ((_hook).he_interested) {                                    \
3079                 hook_pkt_event_t info;                                  \
3080                                                                         \
3081                 _NOTE(CONSTCOND)                                        \
3082                 ASSERT((_ilp != NULL) || (_olp != NULL));               \
3083                                                                         \
3084                 FW_SET_ILL_INDEX(info.hpe_ifp, (ill_t *)_ilp);          \
3085                 FW_SET_ILL_INDEX(info.hpe_ofp, (ill_t *)_olp);          \
3086                 info.hpe_protocol = ipst->ips_ipv4_net_data;            \
3087                 info.hpe_hdr = _iph;                                    \
3088                 info.hpe_mp = &(_fm);                                   \
3089                 info.hpe_mb = _m;                                       \
3090                 info.hpe_flags = _llm;                                  \
3091                 _err = hook_run(ipst->ips_ipv4_net_data->netd_hooks,    \
3092                     _event, (hook_data_t)&info);                        \
3093                 if (_err != 0) {                                        \
3094                         ip2dbg(("%s hook dropped mblk chain %p hdr %p\n",\
3095                             (_hook).he_name, (void *)_fm, (void *)_m)); \
3096                         if (_fm != NULL) {                              \
3097                                 freemsg(_fm);                           \
```



```
3098                                 _fm = NULL;                             \
3099                         }                                               \
3100                         _iph = NULL;                                    \
3101                         _m = NULL;                                      \
3102                 } else {                                                \
3103                         _iph = info.hpe_hdr;                            \
3104                         _m = info.hpe_mb;                               \
3105                 }                                                       \
3106         }

3108 #define FW_HOOKS6(_hook, _event, _ilp, _olp, _iph, _fm, _m, _llm, ipst, _err) \
3109         if ((_hook).he_interested) {                                    \
3110                 hook_pkt_event_t info;                                  \
3111                                                                         \
3112                 _NOTE(CONSTCOND)                                        \
3113                 ASSERT((_ilp != NULL) || (_olp != NULL));               \
3114                                                                         \
3115                 FW_SET_ILL_INDEX(info.hpe_ifp, (ill_t *)_ilp);          \
3116                 FW_SET_ILL_INDEX(info.hpe_ofp, (ill_t *)_olp);          \
3117                 info.hpe_protocol = ipst->ips_ipv6_net_data;            \
3118                 info.hpe_hdr = _iph;                                    \
3119                 info.hpe_mp = &(_fm);                                   \
3120                 info.hpe_mb = _m;                                       \
3121                 info.hpe_flags = _llm;                                  \
3122                 _err = hook_run(ipst->ips_ipv6_net_data->netd_hooks,    \
3123                     _event, (hook_data_t)&info);                        \
3124                 if (_err != 0) {                                        \
3125                         ip2dbg(("%s hook dropped mblk chain %p hdr %p\n",\
3126                             (_hook).he_name, (void *)_fm, (void *)_m)); \
3127                         if (_fm != NULL) {                              \
3128                                 freemsg(_fm);                           \
3129                                 _fm = NULL;                             \
3130                         }                                               \
3131                         _iph = NULL;                                    \
3132                         _m = NULL;                                      \
3133                 } else {                                                \
3134                         _iph = info.hpe_hdr;                            \
3135                         _m = info.hpe_mb;                               \
3136                 }                                                       \
3137         }

3139 #define FW_SET_ILL_INDEX(fp, ill)                                       \
3140         _NOTE(CONSTCOND)                                                \
3141         if ((ill) == NULL || (ill)->ill_phyint == NULL) {               \
3142                 (fp) = 0;                                               \
3143                 _NOTE(CONSTCOND)                                        \
3144         } else if (IS_UNDER_IPMP(ill)) {                                \
3145                 (fp) = ipmp_ill_get_ipmp_ifindex(ill);                  \
3146         } else {                                                        \
3147                 (fp) = (ill)->ill_phyint->phyint_ifindex;               \
3148         }

3150 /*
3151  * Network byte order macros
3152  */
3153 #ifdef  _BIG_ENDIAN
3154 #define N_IN_CLASSA_NET         IN_CLASSA_NET
3155 #define N_IN_CLASSD_NET         IN_CLASSD_NET
3156 #define N_INADDR_UNSPEC_GROUP   INADDR_UNSPEC_GROUP
3157 #define N_IN_LOOPBACK_NET       (ipaddr_t)0x7f000000U
3158 #else /* _BIG_ENDIAN */
3159 #define N_IN_CLASSA_NET         (ipaddr_t)0x000000ffU
3160 #define N_IN_CLASSD_NET         (ipaddr_t)0x000000f0U
3161 #define N_INADDR_UNSPEC_GROUP   (ipaddr_t)0x000000e0U
3162 #define N_IN_LOOPBACK_NET       (ipaddr_t)0x0000007fU
3163 #endif /* _BIG_ENDIAN */
```

```
#define CLASSD(addr)    (((addr) & N_IN_CLASSD_NET) == N_INADDR_UNSPEC_GROUP)
#define CLASSE(addr)    (((addr) & N_IN_CLASSD_NET) == N_IN_CLASSD_NET)
#define IP_LOOPBACK_ADDR(addr)                          \
        (((addr) & N_IN_CLASSA_NET == N_IN_LOOPBACK_NET))

extern int      ip_debug;
extern uint_t   ip_thread_data;
extern krwlock_t ip_thread_rwlock;
extern list_t   ip_thread_list;

#ifdef IP_DEBUG
#include <sys/debug.h>
#include <sys/promif.h>

#define ip0dbg(a)       printf a
#define ip1dbg(a)       if (ip_debug > 2) printf a
#define ip2dbg(a)       if (ip_debug > 3) printf a
#define ip3dbg(a)       if (ip_debug > 4) printf a
#else
#define ip0dbg(a)       /* */
#define ip1dbg(a)       /* */
#define ip2dbg(a)       /* */
#define ip3dbg(a)       /* */
#endif  /* IP_DEBUG */

/* Default MAC-layer address string length for mac_colon_addr */
#define MAC_STR_LEN     128

struct  mac_header_info_s;

extern void     ill_frag_timer(void *);
extern ill_t    *ill_first(int, int, ill_walk_context_t *, ip_stack_t *);
extern ill_t    *ill_next(ill_walk_context_t *, ill_t *);
extern void     ill_frag_timer_start(ill_t *);
extern void     ill_nic_event_dispatch(ill_t *, lif_if_t, nic_event_t,
    nic_event_data_t, size_t);
extern mblk_t   *ip_carve_mp(mblk_t **, ssize_t);
extern mblk_t   *ip_dlpi_alloc(size_t, t_uscalar_t);
extern mblk_t   *ip_dlnotify_alloc(uint_t, uint_t);
extern mblk_t   *ip_dlnotify_alloc2(uint_t, uint_t, uint_t);
extern char     *ip_dot_addr(ipaddr_t, char *);
extern const char *mac_colon_addr(const uint8_t *, size_t, char *, size_t);
extern void     ip_lwput(queue_t *, mblk_t *);
extern boolean_t icmp_err_rate_limit(ip_stack_t *);
extern void     icmp_frag_needed(mblk_t *, int, ip_recv_attr_t *);
extern mblk_t   *icmp_inbound_v4(mblk_t *, ip_recv_attr_t *);
extern void     icmp_time_exceeded(mblk_t *, uint8_t, ip_recv_attr_t *);
extern void     icmp_unreachable(mblk_t *, uint8_t, ip_recv_attr_t *);
extern boolean_t ip_ipsec_policy_inherit(conn_t *, conn_t *, ip_recv_attr_t *);
extern void     *ip_pullup(mblk_t *, ssize_t, ip_recv_attr_t *);
extern void     ip_setl2src(mblk_t *, ip_recv_attr_t *, ill_t *);
extern mblk_t   *ip_check_and_align_header(mblk_t *, uint_t, ip_recv_attr_t *);
extern mblk_t   *ip_check_length(mblk_t *, uchar_t *, ssize_t, uint_t, uint_t,
    ip_recv_attr_t *);
extern mblk_t   *ip_check_optlen(mblk_t *, ipha_t *, uint_t, uint_t,
    ip_recv_attr_t *);
extern mblk_t   *ip_fix_dbref(mblk_t *, ip_recv_attr_t *);
extern uint_t   ip_cksum(mblk_t *, int, uint32_t);
extern int      ip_close(queue_t *, int);
extern uint16_t ip_csum_hdr(ipha_t *);
extern void     ip_forward_xmit_v4(nce_t *, ill_t *, mblk_t *, ipha_t *,
    ip_recv_attr_t *, uint32_t, uint32_t);
extern boolean_t ip_forward_options(mblk_t *, ipha_t *, ill_t *,
    ip_recv_attr_t *);
extern int      ip_fragment_v4(mblk_t *, nce_t *, iaflags_t, uint_t, uint32_t,
    uint32_t, zoneid_t, zoneid_t, pfirepostfrag_t postfragfn,
```



```
    uintptr_t *cookie);
extern void     ip_proto_not_sup(mblk_t *, ip_recv_attr_t *);
extern void     ip_ire_g_fini(void);
extern void     ip_ire_g_init(void);
extern void     ip_ire_fini(ip_stack_t *);
extern void     ip_ire_init(ip_stack_t *);
extern void     ip_mdata_to_mhi(ill_t *, mblk_t *, struct mac_header_info_s *);
extern int      ip_openv4(queue_t *q, dev_t *devp, int flag, int sflag,
                    cred_t *credp);
extern int      ip_openv6(queue_t *q, dev_t *devp, int flag, int sflag,
                    cred_t *credp);
extern int      ip_reassemble(mblk_t *, ipf_t *, uint_t, boolean_t, ill_t *,
    size_t);
extern void     ip_rput(queue_t *, mblk_t *);
extern void     ip_input(ill_t *, ill_rx_ring_t *, mblk_t *,
    struct mac_header_info_s *);
extern void     ip_input_v6(ill_t *, ill_rx_ring_t *, mblk_t *,
    struct mac_header_info_s *);
extern mblk_t   *ip_input_common_v4(ill_t *, ill_rx_ring_t *, mblk_t *,
    struct mac_header_info_s *, squeue_t *, mblk_t **, uint_t *);
extern mblk_t   *ip_input_common_v6(ill_t *, ill_rx_ring_t *, mblk_t *,
    struct mac_header_info_s *, squeue_t *, mblk_t **, uint_t *);
extern void     ill_input_full_v4(mblk_t *, void *, void *,
    ip_recv_attr_t *, rtc_t *);
extern void     ill_input_short_v4(mblk_t *, void *, void *,
    ip_recv_attr_t *, rtc_t *);
extern void     ill_input_full_v6(mblk_t *, void *, void *,
    ip_recv_attr_t *, rtc_t *);
extern void     ill_input_short_v6(mblk_t *, void *, void *,
    ip_recv_attr_t *, rtc_t *);
extern ipaddr_t ip_input_options(ipha_t *, ipaddr_t, mblk_t *,
    ip_recv_attr_t *, int *);
extern boolean_t ip_input_local_options(mblk_t *, ipha_t *, ip_recv_attr_t *);
extern mblk_t   *ip_input_fragment(mblk_t *, ipha_t *, ip_recv_attr_t *);
extern mblk_t   *ip_input_fragment_v6(mblk_t *, ip6_t *, ip6_frag_t *, uint_t,
    ip_recv_attr_t *);
extern void     ip_input_post_ipsec(mblk_t *, ip_recv_attr_t *);
extern void     ip_fanout_v4(mblk_t *, ipha_t *, ip_recv_attr_t *);
extern void     ip_fanout_v6(mblk_t *, ip6_t *, ip_recv_attr_t *);
extern void     ip_fanout_proto_conn(conn_t *, mblk_t *, ipha_t *, ip6_t *,
    ip_recv_attr_t *);
extern void     ip_fanout_proto_v4(mblk_t *, ipha_t *, ip_recv_attr_t *);
extern void     ip_fanout_send_icmp_v4(mblk_t *, uint_t, uint_t,
    ip_recv_attr_t *);
extern void     ip_fanout_udp_conn(conn_t *, mblk_t *, ipha_t *, ip6_t *,
    ip_recv_attr_t *);
extern void     ip_fanout_udp_multi_v4(mblk_t *, ipha_t *, uint16_t, uint16_t,
    ip_recv_attr_t *);
extern mblk_t   *zero_spi_check(mblk_t *, ip_recv_attr_t *);
extern void     ip_build_hdrs_v4(uchar_t *, uint_t, const ip_pkt_t *, uint8_t);
extern int      ip_find_hdr_v4(ipha_t *, ip_pkt_t *, boolean_t);
extern int      ip_total_hdrs_len_v4(const ip_pkt_t *);

extern mblk_t   *ip_accept_tcp(ill_t *, ill_rx_ring_t *, squeue_t *,
    mblk_t *, mblk_t **, uint_t *cnt);
extern void     ip_rput_dlpi(ill_t *, mblk_t *);
extern void     ip_rput_notdata(ill_t *, mblk_t *);

extern void     ip_mib2_add_ip_stats(mib2_ipIfStatsEntry_t *,
                    mib2_ipIfStatsEntry_t *);
extern void     ip_mib2_add_icmp6_stats(mib2_ipv6IfIcmpEntry_t *,
                    mib2_ipv6IfIcmpEntry_t *);
extern void     ip_rput_other(ipsq_t *, queue_t *, mblk_t *, void *);
extern ire_t    *ip_check_multihome(void *, ire_t *, ill_t *);
extern void     ip_send_potential_redirect_v4(mblk_t *, ipha_t *, ire_t *,
    ip_recv_attr_t *);
```

```
3296 extern int      ip_set_destination_v4(ipaddr_t *, ipaddr_t, ipaddr_t,
3297     ip_xmit_attr_t *, iulp_t *, uint32_t, uint_t);
3298 extern int      ip_set_destination_v6(in6_addr_t *, const in6_addr_t *,
3299     const in6_addr_t *, ip_xmit_attr_t *, iulp_t *, uint32_t, uint_t);

3301 extern int      ip_output_simple(mblk_t *, ip_xmit_attr_t *);
3302 extern int      ip_output_simple_v4(mblk_t *, ip_xmit_attr_t *);
3303 extern int      ip_output_simple_v6(mblk_t *, ip_xmit_attr_t *);
3304 extern int      ip_output_options(mblk_t *, ipha_t *, ip_xmit_attr_t *,
3305     ill_t *);
3306 extern void     ip_output_local_options(ipha_t *, ip_stack_t *);

3308 extern ip_xmit_attr_t *conn_get_ixa(conn_t *, boolean_t);
3309 extern ip_xmit_attr_t *conn_get_ixa_tryhard(conn_t *, boolean_t);
3310 extern ip_xmit_attr_t *conn_replace_ixa(conn_t *, ip_xmit_attr_t *);
3311 extern ip_xmit_attr_t *conn_get_ixa_exclusive(conn_t *);
3312 extern ip_xmit_attr_t *ip_xmit_attr_duplicate(ip_xmit_attr_t *);
3313 extern void     ip_xmit_attr_replace_tsl(ip_xmit_attr_t *, ts_label_t *);
3314 extern void     ip_xmit_attr_restore_tsl(ip_xmit_attr_t *, cred_t *);
3315 boolean_t       ip_recv_attr_replace_label(ip_recv_attr_t *, ts_label_t *);
3316 extern void     ixa_inactive(ip_xmit_attr_t *);
3317 extern void     ixa_refrele(ip_xmit_attr_t *);
3318 extern boolean_t ixa_check_drain_insert(conn_t *, ip_xmit_attr_t *);
3319 extern void     ixa_cleanup(ip_xmit_attr_t *);
3320 extern void     ira_cleanup(ip_recv_attr_t *, boolean_t);
3321 extern void     ixa_safe_copy(ip_xmit_attr_t *, ip_xmit_attr_t *);

3323 extern int      conn_ip_output(mblk_t *, ip_xmit_attr_t *);
3324 extern boolean_t ip_output_verify_local(ip_xmit_attr_t *);
3325 extern mblk_t   *ip_output_process_local(mblk_t *, ip_xmit_attr_t *, boolean_t,
3326     boolean_t, conn_t *);

3328 extern int      conn_opt_get(conn_opt_arg_t *, t_scalar_t, t_scalar_t,
3329     uchar_t *);
3330 extern int      conn_opt_set(conn_opt_arg_t *, t_scalar_t, t_scalar_t, uint_t,
3331     uchar_t *, boolean_t, cred_t *);
3332 extern boolean_t        conn_same_as_last_v4(conn_t *, sin_t *);
3333 extern boolean_t        conn_same_as_last_v6(conn_t *, sin6_t *);
3334 extern int      conn_update_label(const conn_t *, const ip_xmit_attr_t *,
3335     const in6_addr_t *, ip_pkt_t *);

3337 extern int      ip_opt_set_multicast_group(conn_t *, t_scalar_t,
3338     uchar_t *, boolean_t, boolean_t);
3339 extern int      ip_opt_set_multicast_sources(conn_t *, t_scalar_t,
3340     uchar_t *, boolean_t, boolean_t);
3341 extern int      conn_getsockname(conn_t *, struct sockaddr *, uint_t *);
3342 extern int      conn_getpeername(conn_t *, struct sockaddr *, uint_t *);

3344 extern int      conn_build_hdr_template(conn_t *, uint_t, uint_t,
3345     const in6_addr_t *, const in6_addr_t *, uint32_t);
3346 extern mblk_t   *conn_prepend_hdr(ip_xmit_attr_t *, const ip_pkt_t *,
3347     const in6_addr_t *, const in6_addr_t *, uint8_t, uint32_t, uint_t,
3348     mblk_t *, uint_t, uint_t, uint32_t *, int *);
3349 extern void     ip_attr_newdst(ip_xmit_attr_t *);
3350 extern void     ip_attr_nexthop(const ip_pkt_t *, const ip_xmit_attr_t *,
3351     const in6_addr_t *, in6_addr_t *);
3352 extern int      conn_connect(conn_t *, iulp_t *, uint32_t);
3353 extern int      ip_attr_connect(const conn_t *, ip_xmit_attr_t *,
3354     const in6_addr_t *, const in6_addr_t *, const in6_addr_t *, in_port_t,
3355     in6_addr_t *, iulp_t *, uint32_t);
3356 extern int      conn_inherit_parent(conn_t *, conn_t *);

3358 extern void     conn_ixa_cleanup(conn_t *connp, void *arg);

3360 extern boolean_t conn_wantpacket(conn_t *, ip_recv_attr_t *, ipha_t *);
3361 extern uint_t   ip_type_v4(ipaddr_t, ip_stack_t *);
```



```
3362 extern uint_t   ip_type_v6(const in6_addr_t *, ip_stack_t *);

3364 extern void     ip_wput_nondata(queue_t *, mblk_t *);
3365 extern void     ip_wsrv(queue_t *);
3366 extern char     *ip_nv_lookup(nv_t *, int);
3367 extern boolean_t ip_local_addr_ok_v6(const in6_addr_t *, const in6_addr_t *);
3368 extern boolean_t ip_remote_addr_ok_v6(const in6_addr_t *, const in6_addr_t *);
3369 extern ipaddr_t ip_massage_options(ipha_t *, netstack_t *);
3370 extern ipaddr_t ip_net_mask(ipaddr_t);
3371 extern void     arp_bringup_done(ill_t *, int);
3372 extern void     arp_replumb_done(ill_t *, int);

3374 extern struct qinit iprinitv6;

3376 extern void     ipmp_init(ip_stack_t *);
3377 extern void     ipmp_destroy(ip_stack_t *);
3378 extern ipmp_grp_t *ipmp_grp_create(const char *, phyint_t *);
3379 extern void     ipmp_grp_destroy(ipmp_grp_t *);
3380 extern void     ipmp_grp_info(const ipmp_grp_t *, lifgroupinfo_t *);
3381 extern int      ipmp_grp_rename(ipmp_grp_t *, const char *);
3382 extern ipmp_grp_t *ipmp_grp_lookup(const char *, ip_stack_t *);
3383 extern int      ipmp_grp_vet_phyint(ipmp_grp_t *, phyint_t *);
3384 extern ipmp_illgrp_t *ipmp_illgrp_create(ill_t *);
3385 extern void     ipmp_illgrp_destroy(ipmp_illgrp_t *);
3386 extern ill_t    *ipmp_illgrp_add_ipif(ipmp_illgrp_t *, ipif_t *);
3387 extern void     ipmp_illgrp_del_ipif(ipmp_illgrp_t *, ipif_t *);
3388 extern ill_t    *ipmp_illgrp_next_ill(ipmp_illgrp_t *);
3389 extern ill_t    *ipmp_illgrp_hold_next_ill(ipmp_illgrp_t *);
3390 extern ill_t    *ipmp_illgrp_hold_cast_ill(ipmp_illgrp_t *);
3391 extern ill_t    *ipmp_illgrp_ipmp_ill(ipmp_illgrp_t *);
3392 extern void     ipmp_illgrp_refresh_mtu(ipmp_illgrp_t *);
3393 extern ipmp_arpent_t *ipmp_illgrp_create_arpent(ipmp_illgrp_t *,
3394     boolean_t, ipaddr_t, uchar_t *, size_t, uint16_t);
3395 extern void     ipmp_illgrp_destroy_arpent(ipmp_illgrp_t *, ipmp_arpent_t *);
3396 extern ipmp_arpent_t *ipmp_illgrp_lookup_arpent(ipmp_illgrp_t *, ipaddr_t *);
3397 extern void     ipmp_illgrp_refresh_arpent(ipmp_illgrp_t *);
3398 extern void     ipmp_illgrp_mark_arpent(ipmp_illgrp_t *, ipmp_arpent_t *);
3399 extern ill_t    *ipmp_illgrp_find_ill(ipmp_illgrp_t *, uchar_t *, uint_t);
3400 extern void     ipmp_illgrp_link_grp(ipmp_illgrp_t *, ipmp_grp_t *);
3401 extern int      ipmp_illgrp_unlink_grp(ipmp_illgrp_t *);
3402 extern uint_t   ipmp_ill_get_ipmp_ifindex(const ill_t *);
3403 extern void     ipmp_ill_join_illgrp(ill_t *, ipmp_illgrp_t *);
3404 extern void     ipmp_ill_leave_illgrp(ill_t *);
3405 extern ill_t    *ipmp_ill_hold_ipmp_ill(ill_t *);
3406 extern ill_t    *ipmp_ill_hold_xmit_ill(ill_t *, boolean_t);
3407 extern boolean_t ipmp_ill_is_active(ill_t *);
3408 extern void     ipmp_ill_refresh_active(ill_t *);
3409 extern void     ipmp_phyint_join_grp(phyint_t *, ipmp_grp_t *);
3410 extern void     ipmp_phyint_leave_grp(phyint_t *);
3411 extern void     ipmp_phyint_refresh_active(phyint_t *);
3412 extern ill_t    *ipmp_ipif_bound_ill(const ipif_t *);
3413 extern ill_t    *ipmp_ipif_hold_bound_ill(const ipif_t *);
3414 extern boolean_t ipmp_ipif_is_dataaddr(const ipif_t *);
3415 extern boolean_t ipmp_ipif_is_stubaddr(const ipif_t *);
3416 extern boolean_t ipmp_packet_is_probe(mblk_t *, ill_t *);
3417 extern void     ipmp_ncec_delete_nce(ncec_t *);
3418 extern void     ipmp_ncec_refresh_nce(ncec_t *);

3420 extern void     conn_drain_insert(conn_t *, idl_tx_list_t *);
3421 extern void     conn_setqfull(conn_t *, boolean_t *);
3422 extern void     conn_clrqfull(conn_t *, boolean_t *);
3423 extern int      conn_ipsec_length(conn_t *);
3424 extern ipaddr_t ip_get_dst(ipha_t *);
3425 extern uint_t   ip_get_pmtu(ip_xmit_attr_t *);
3426 extern uint_t   ip_get_base_mtu(ill_t *, ire_t *);
3427 extern mblk_t *ip_output_attach_policy(mblk_t *, ipha_t *, ip6_t *,
```

```
3428     const conn_t *, ip_xmit_attr_t *);
3429 extern int      ipsec_out_extra_length(ip_xmit_attr_t *);
3430 extern int      ipsec_out_process(mblk_t *, ip_xmit_attr_t *);
3431 extern int      ip_output_post_ipsec(mblk_t *, ip_xmit_attr_t *);
3432 extern void     ipsec_out_to_in(ip_xmit_attr_t *, ill_t *ill,
3433     ip_recv_attr_t *);

3435 extern void     ire_cleanup(ire_t *);
3436 extern void     ire_inactive(ire_t *);
3437 extern boolean_t irb_inactive(irb_t *);
3438 extern ire_t    *ire_unlink(irb_t *);

3440 #ifdef DEBUG
3441 extern  boolean_t th_trace_ref(const void *, ip_stack_t *);
3442 extern  void    th_trace_unref(const void *);
3443 extern  void    th_trace_cleanup(const void *, boolean_t);
3444 extern  void    ire_trace_ref(ire_t *);
3445 extern  void    ire_untrace_ref(ire_t *);
3446 #endif

3448 extern int      ip_srcid_insert(const in6_addr_t *, zoneid_t, ip_stack_t *);
3449 extern int      ip_srcid_remove(const in6_addr_t *, zoneid_t, ip_stack_t *);
3450 extern void     ip_srcid_find_id(uint_t, in6_addr_t *, zoneid_t, netstack_t *);
3451 extern uint_t   ip_srcid_find_addr(const in6_addr_t *, zoneid_t, netstack_t *);

3453 extern uint8_t  ipoptp_next(ipoptp_t *);
3454 extern uint8_t  ipoptp_first(ipoptp_t *, ipha_t *);
3455 extern int      ip_opt_get_user(conn_t *, uchar_t *);
3456 extern int      ipsec_req_from_conn(conn_t *, ipsec_req_t *, int);
3457 extern int      ip_snmp_get(queue_t *q, mblk_t *mctl, int level, boolean_t);
3458 extern int      ip_snmp_set(queue_t *q, int, int, uchar_t *, int);
3459 extern void     ip_process_ioctl(ipsq_t *, queue_t *, mblk_t *, void *);
3460 extern void     ip_quiesce_conn(conn_t *);
3461 extern  void    ip_reprocess_ioctl(ipsq_t *, queue_t *, mblk_t *, void *);
3462 extern void     ip_ioctl_finish(queue_t *, mblk_t *, int, int, ipsq_t *);

3464 extern boolean_t ip_cmpbuf(const void *, uint_t, boolean_t, const void *,
3465     uint_t);
3466 extern boolean_t ip_allocbuf(void **, uint_t *, boolean_t, const void *,
3467     uint_t);
3468 extern void     ip_savebuf(void **, uint_t *, boolean_t, const void *, uint_t);

3470 extern boolean_t        ipsq_pending_mp_cleanup(ill_t *, conn_t *);
3471 extern void     conn_ioctl_cleanup(conn_t *);

3473 extern void     ip_unbind(conn_t *);

3475 extern void tnet_init(void);
3476 extern void tnet_fini(void);

3478 /*
3479  * Hook functions to enable cluster networking
3480  * On non-clustered systems these vectors must always be NULL.
3481  */
3482 extern int (*cl_inet_isclusterwide)(netstackid_t stack_id, uint8_t protocol,
3483     sa_family_t addr_family, uint8_t *laddrp, void *args);
3484 extern uint32_t (*cl_inet_ipident)(netstackid_t stack_id, uint8_t protocol,
3485     sa_family_t addr_family, uint8_t *laddrp, uint8_t *faddrp,
3486     void *args);
3487 extern int (*cl_inet_connect2)(netstackid_t stack_id, uint8_t protocol,
3488     boolean_t is_outgoing, sa_family_t addr_family, uint8_t *laddrp,
3489     in_port_t lport, uint8_t *faddrp, in_port_t fport, void *args);
3490 extern void (*cl_inet_getspi)(netstackid_t, uint8_t, uint8_t *, size_t,
3491     void *);
3492 extern void (*cl_inet_getspi)(netstackid_t stack_id, uint8_t protocol,
3493     uint8_t *ptr, size_t len, void *args);
```



```
3494 extern int (*cl_inet_checkspi)(netstackid_t stack_id, uint8_t protocol,
3495     uint32_t spi, void *args);
3496 extern void (*cl_inet_deletespi)(netstackid_t stack_id, uint8_t protocol,
3497     uint32_t spi, void *args);
3498 extern void (*cl_inet_idlesa)(netstackid_t, uint8_t, uint32_t,
3499     sa_family_t, in6_addr_t, in6_addr_t, void *);


3502 /* Hooks for CGTP (multirt routes) filtering module */
3503 #define CGTP_FILTER_REV_1       1
3504 #define CGTP_FILTER_REV_2       2
3505 #define CGTP_FILTER_REV_3       3
3506 #define CGTP_FILTER_REV         CGTP_FILTER_REV_3

3508 /* cfo_filter and cfo_filter_v6 hooks return values */
3509 #define CGTP_IP_PKT_NOT_CGTP    0
3510 #define CGTP_IP_PKT_PREMIUM     1
3511 #define CGTP_IP_PKT_DUPLICATE   2

3513 /* Version 3 of the filter interface */
3514 typedef struct cgtp_filter_ops {
3515         int     cfo_filter_rev;                 /* CGTP_FILTER_REV_3 */
3516         int     (*cfo_change_state)(netstackid_t, int);
3517         int     (*cfo_add_dest_v4)(netstackid_t, ipaddr_t, ipaddr_t,
3518                     ipaddr_t, ipaddr_t);
3519         int     (*cfo_del_dest_v4)(netstackid_t, ipaddr_t, ipaddr_t);
3520         int     (*cfo_add_dest_v6)(netstackid_t, in6_addr_t *, in6_addr_t *,
3521                     in6_addr_t *, in6_addr_t *);
3522         int     (*cfo_del_dest_v6)(netstackid_t, in6_addr_t *, in6_addr_t *);
3523         int     (*cfo_filter)(netstackid_t, uint_t, mblk_t *);
3524         int     (*cfo_filter_v6)(netstackid_t, uint_t, ip6_t *,
3525                     ip6_frag_t *);
3526 } cgtp_filter_ops_t;

3528 #define CGTP_MCAST_SUCCESS      1

3530 /*
3531  * The separate CGTP module needs this global symbol so that it
3532  * can check the version and determine whether to use the old or the new
3533  * version of the filtering interface.
3534  */
3535 extern int      ip_cgtp_filter_rev;

3537 extern int      ip_cgtp_filter_supported(void);
3538 extern int      ip_cgtp_filter_register(netstackid_t, cgtp_filter_ops_t *);
3539 extern int      ip_cgtp_filter_unregister(netstackid_t);
3540 extern int      ip_cgtp_filter_is_registered(netstackid_t);

3542 /*
3543  * rr_ring_state cycles in the order shown below from RR_FREE through
3544  * RR_FREE_IN_PROG and  back to RR_FREE.
3545  */
3546 typedef enum {
3547         RR_FREE,                        /* Free slot */
3548         RR_SQUEUE_UNBOUND,              /* Ring's squeue is unbound */
3549         RR_SQUEUE_BIND_INPROG,          /* Ring's squeue bind in progress */
3550         RR_SQUEUE_BOUND,                /* Ring's squeue bound to cpu */
3551         RR_FREE_INPROG                  /* Ring is being freed */
3552 } ip_ring_state_t;

3554 #define ILL_MAX_RINGS           256     /* Max num of rx rings we can manage */
3555 #define ILL_POLLING             0x01    /* Polling in use */

3557 /*
3558  * These functions pointer types are exported by the mac/dls layer.
3559  * we need to duplicate the definitions here because we cannot
```

```
 * include mac/dls header files here.
 */
typedef boolean_t               (*ip_mac_intr_disable_t)(void *);
typedef void                    (*ip_mac_intr_enable_t)(void *);
typedef ip_mac_tx_cookie_t      (*ip_dld_tx_t)(void *, mblk_t *,
    uint64_t, uint16_t);
typedef void                    (*ip_flow_enable_t)(void *, ip_mac_tx_cookie_t);
typedef void                    *(*ip_dld_callb_t)(void *,
    ip_flow_enable_t, void *);
typedef boolean_t               (*ip_dld_fctl_t)(void *, ip_mac_tx_cookie_t);
typedef int                     (*ip_capab_func_t)(void *, uint_t,
    void *, uint_t);

/*
 * POLLING README
 * sq_get_pkts() is called to pick packets from softring in poll mode. It
 * calls rr_rx to get the chain and process it with rr_ip_accept.
 * rr_rx = mac_soft_ring_poll() to pick packets
 * rr_ip_accept = ip_accept_tcp() to process packets
 */

/*
 * XXX: With protocol, service specific squeues, they will have
 * specific acceptor functions.
 */
typedef mblk_t *(*ip_mac_rx_t)(void *, size_t);
typedef mblk_t *(*ip_accept_t)(ill_t *, ill_rx_ring_t *,
    squeue_t *, mblk_t *, mblk_t **, uint_t *);

/*
 * rr_intr_enable, rr_intr_disable, rr_rx_handle, rr_rx:
 * May be accessed while in the squeue AND after checking that SQS_POLL_CAPAB
 * is set.
 *
 * rr_ring_state: Protected by ill_lock.
 */
struct ill_rx_ring {
        ip_mac_intr_disable_t   rr_intr_disable; /* Interrupt disabling func */
        ip_mac_intr_enable_t    rr_intr_enable; /* Interrupt enabling func */
        void                    *rr_intr_handle; /* Handle interrupt funcs */
        ip_mac_rx_t             rr_rx;          /* Driver receive function */
        ip_accept_t             rr_ip_accept;   /* IP accept function */
        void                    *rr_rx_handle;  /* Handle for Rx ring */
        squeue_t                *rr_sqp; /* Squeue the ring is bound to */
        ill_t                   *rr_ill;        /* back pointer to ill */
        ip_ring_state_t         rr_ring_state;  /* State of this ring */
};

/*
 * IP - DLD direct function call capability
 * Suffixes, df - dld function, dh - dld handle,
 * cf - client (IP) function, ch - client handle
 */
typedef struct ill_dld_direct_s {               /* DLD provided driver Tx */
        ip_dld_tx_t             idd_tx_df;      /* str_mdata_fastpath_put */
        void                    *idd_tx_dh;     /* dld_str_t *dsp */
        ip_dld_callb_t          idd_tx_cb_df;   /* mac_tx_srs_notify */
        void                    *idd_tx_cb_dh;  /* mac_client_handle_t *mch */
        ip_dld_fctl_t           idd_tx_fctl_df; /* mac_tx_is_flow_blocked */
        void                    *idd_tx_fctl_dh;        /* mac_client_handle */
} ill_dld_direct_t;

/* IP - DLD polling capability */
typedef struct ill_dld_poll_s {
        ill_rx_ring_t           idp_ring_tbl[ILL_MAX_RINGS];
} ill_dld_poll_t;
```



```
/* Describes ill->ill_dld_capab */
struct ill_dld_capab_s {
        ip_capab_func_t         idc_capab_df;   /* dld_capab_func */
        void                    *idc_capab_dh;  /* dld_str_t *dsp */
        ill_dld_direct_t        idc_direct;
        ill_dld_poll_t          idc_poll;
};

/*
 * IP squeues exports
 */
extern boolean_t        ip_squeue_fanout;

#define IP_SQUEUE_GET(hint) ip_squeue_random(hint)

extern void ip_squeue_init(void (*)(squeue_t *));
extern squeue_t *ip_squeue_random(uint_t);
extern squeue_t *ip_squeue_get(ill_rx_ring_t *);
extern squeue_t *ip_squeue_getfree(pri_t);
extern int ip_squeue_cpu_move(squeue_t *, processorid_t);
extern void *ip_squeue_add_ring(ill_t *, void *);
extern void ip_squeue_bind_ring(ill_t *, ill_rx_ring_t *, processorid_t);
extern void ip_squeue_clean_ring(ill_t *, ill_rx_ring_t *);
extern void ip_squeue_quiesce_ring(ill_t *, ill_rx_ring_t *);
extern void ip_squeue_restart_ring(ill_t *, ill_rx_ring_t *);
extern void ip_squeue_clean_all(ill_t *);
extern boolean_t        ip_source_routed(ipha_t *, ip_stack_t *);

extern void tcp_wput(queue_t *, mblk_t *);

extern int      ip_fill_mtuinfo(conn_t *, ip_xmit_attr_t *,
    struct ip6_mtuinfo *);
extern hook_t *ipobs_register_hook(netstack_t *, pfv_t);
extern void ipobs_unregister_hook(netstack_t *, hook_t *);
extern void ipobs_hook(mblk_t *, int, zoneid_t, zoneid_t, const ill_t *,
    ip_stack_t *);
typedef void    (*ipsq_func_t)(ipsq_t *, queue_t *, mblk_t *, void *);

extern void     dce_g_init(void);
extern void     dce_g_destroy(void);
extern void     dce_stack_init(ip_stack_t *);
extern void     dce_stack_destroy(ip_stack_t *);
extern void     dce_cleanup(uint_t, ip_stack_t *);
extern dce_t    *dce_get_default(ip_stack_t *);
extern dce_t    *dce_lookup_pkt(mblk_t *, ip_xmit_attr_t *, uint_t *);
extern dce_t    *dce_lookup_v4(ipaddr_t, ip_stack_t *, uint_t *);
extern dce_t    *dce_lookup_v6(const in6_addr_t *, uint_t, ip_stack_t *,
    uint_t *);
extern dce_t    *dce_lookup_and_add_v4(ipaddr_t, ip_stack_t *);
extern dce_t    *dce_lookup_and_add_v6(const in6_addr_t *, uint_t,
    ip_stack_t *);
extern int      dce_update_uinfo_v4(ipaddr_t, iulp_t *, ip_stack_t *);
extern int      dce_update_uinfo_v6(const in6_addr_t *, uint_t, iulp_t *,
    ip_stack_t *);
extern int      dce_update_uinfo(const in6_addr_t *, uint_t, iulp_t *,
    ip_stack_t *);
extern void     dce_increment_generation(dce_t *);
extern void     dce_increment_all_generations(boolean_t, ip_stack_t *);
extern void     dce_refrele(dce_t *);
extern void     dce_refhold(dce_t *);
extern void     dce_refrele_notr(dce_t *);
extern void     dce_refhold_notr(dce_t *);
mblk_t          *ip_snmp_get_mib2_ip_dce(queue_t *, mblk_t *, ip_stack_t *ipst);

extern ip_laddr_t ip_laddr_verify_v4(ipaddr_t, zoneid_t,
```

```
3692     ip_stack_t *, boolean_t);
3693 extern ip_laddr_t ip_laddr_verify_v6(const in6_addr_t *, zoneid_t,
3694     ip_stack_t *, boolean_t, uint_t);
3695 extern int      ip_laddr_fanout_insert(conn_t *);

3697 extern boolean_t ip_verify_src(mblk_t *, ip_xmit_attr_t *, uint_t *);
3698 extern int      ip_verify_ire(mblk_t *, ip_xmit_attr_t *);

3700 extern mblk_t   *ip_xmit_attr_to_mblk(ip_xmit_attr_t *);
3701 extern boolean_t ip_xmit_attr_from_mblk(mblk_t *, ip_xmit_attr_t *);
3702 extern mblk_t   *ip_xmit_attr_free_mblk(mblk_t *);
3703 extern mblk_t   *ip_recv_attr_to_mblk(ip_recv_attr_t *);
3704 extern boolean_t ip_recv_attr_from_mblk(mblk_t *, ip_recv_attr_t *);
3705 extern mblk_t   *ip_recv_attr_free_mblk(mblk_t *);
3706 extern boolean_t ip_recv_attr_is_mblk(mblk_t *);

3708 /*
3709  * Squeue tags. Tags only need to be unique when the callback function is the
3710  * same to distinguish between different calls, but we use unique tags for
3711  * convenience anyway.
3712  */
3713 #define SQTAG_IP_INPUT                  1
3714 #define SQTAG_TCP_INPUT_ICMP_ERR        2
3715 #define SQTAG_TCP6_INPUT_ICMP_ERR       3
3716 #define SQTAG_IP_TCP_INPUT              4
3717 #define SQTAG_IP6_TCP_INPUT             5
3718 #define SQTAG_IP_TCP_CLOSE              6
3719 #define SQTAG_TCP_OUTPUT                7
3720 #define SQTAG_TCP_TIMER                 8
3721 #define SQTAG_TCP_TIMEWAIT              9
3722 #define SQTAG_TCP_ACCEPT_FINISH         10
3723 #define SQTAG_TCP_ACCEPT_FINISH_Q0      11
3724 #define SQTAG_TCP_ACCEPT_PENDING        12
3725 #define SQTAG_TCP_LISTEN_DISCON         13
3726 #define SQTAG_TCP_CONN_REQ_1            14
3727 #define SQTAG_TCP_EAGER_BLOWOFF         15
3728 #define SQTAG_TCP_EAGER_CLEANUP         16
3729 #define SQTAG_TCP_EAGER_CLEANUP_Q0      17
3730 #define SQTAG_TCP_CONN_IND              18
3731 #define SQTAG_TCP_RSRV                  19
3732 #define SQTAG_TCP_ABORT_BUCKET          20
3733 #define SQTAG_TCP_REINPUT               21
3734 #define SQTAG_TCP_REINPUT_EAGER         22
3735 #define SQTAG_TCP_INPUT_MCTL            23
3736 #define SQTAG_TCP_RPUTOTHER             24
3737 #define SQTAG_IP_PROTO_AGAIN            25
3738 #define SQTAG_IP_FANOUT_TCP             26
3739 #define SQTAG_IPSQ_CLEAN_RING           27
3740 #define SQTAG_TCP_WPUT_OTHER            28
3741 #define SQTAG_TCP_CONN_REQ_UNBOUND      29
3742 #define SQTAG_TCP_SEND_PENDING          30
3743 #define SQTAG_BIND_RETRY                31
3744 #define SQTAG_UDP_FANOUT                32
3745 #define SQTAG_UDP_INPUT                 33
3746 #define SQTAG_UDP_WPUT                  34
3747 #define SQTAG_UDP_OUTPUT                35
3748 #define SQTAG_TCP_KSSL_INPUT            36
3749 #define SQTAG_TCP_DROP_Q0               37
3750 #define SQTAG_TCP_CONN_REQ_2            38
3751 #define SQTAG_IP_INPUT_RX_RING          39
3752 #define SQTAG_SQUEUE_CHANGE             40
3753 #define SQTAG_CONNECT_FINISH            41
3754 #define SQTAG_SYNCHRONOUS_OP            42
3755 #define SQTAG_TCP_SHUTDOWN_OUTPUT       43
3756 #define SQTAG_TCP_IXA_CLEANUP           44
3757 #define SQTAG_TCP_SEND_SYNACK           45
```



```
3758 #define SQTAG_IP_DCCP_INPUT             46
3759 #define SQTAG_DCCP_OUTPUT               47
3760 #define SQTAG_DCCP_CONN_REQ_UNBOUND     48
3761 #endif /* ! codereview */

3763 extern sin_t    sin_null;       /* Zero address for quick clears */
3764 extern sin6_t   sin6_null;      /* Zero address for quick clears */

3766 #endif  /* _KERNEL */

3768 #ifdef  __cplusplus
3769 }
3770 #endif

3772 #endif  /* _INET_IP_H */
```

```
**********************************************************
  448654 Mon Jul  9 14:38:15 2012
new/usr/src/uts/common/inet/ip/ip.c
dccp: starting module template
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 1990 Mentat Inc.
 * Copyright (c) 2011 Joyent, Inc. All rights reserved.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/dlpi.h>
#include <sys/stropts.h>
#include <sys/sysmacros.h>
#include <sys/strsubr.h>
#include <sys/strlog.h>
#include <sys/strsun.h>
#include <sys/zone.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/xti_inet.h>
#include <sys/ddi.h>
#include <sys/suntpi.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/kobj.h>
#include <sys/modctl.h>
#include <sys/atomic.h>
#include <sys/policy.h>
#include <sys/priv.h>
#include <sys/taskq.h>

#include <sys/systm.h>
#include <sys/param.h>
#include <sys/kmem.h>
#include <sys/sdt.h>
#include <sys/socket.h>
#include <sys/vtrace.h>
#include <sys/isa_defs.h>
#include <sys/mac.h>
#include <net/if.h>
#include <net/if_arp.h>
#include <net/route.h>
```



```
#include <sys/sockio.h>
#include <netinet/in.h>
#include <net/if_dl.h>

#include <inet/common.h>
#include <inet/mi.h>
#include <inet/mib2.h>
#include <inet/nd.h>
#include <inet/arp.h>
#include <inet/snmpcom.h>
#include <inet/optcom.h>
#include <inet/kstatcom.h>

#include <netinet/igmp_var.h>
#include <netinet/ip6.h>
#include <netinet/icmp6.h>
#include <netinet/sctp.h>

#include <inet/ip.h>
#include <inet/ip_impl.h>
#include <inet/ip6.h>
#include <inet/ip6_asp.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>
#include <inet/ip_multi.h>
#include <inet/ip_if.h>
#include <inet/ip_ire.h>
#include <inet/ip_ftable.h>
#include <inet/ip_rts.h>
#include <inet/ip_ndp.h>
#include <inet/ip_listutils.h>
#include <netinet/igmp.h>
#include <netinet/ip_mroute.h>
#include <inet/ipp_common.h>

#include <net/pfkeyv2.h>
#include <inet/sadb.h>
#include <inet/ipsec_impl.h>
#include <inet/iptun/iptun_impl.h>
#include <inet/ipdrop.h>
#include <inet/ip_netinfo.h>
#include <inet/ilb_ip.h>

#include <sys/ethernet.h>
#include <net/if_types.h>
#include <sys/cpuvar.h>

#include <ipp/ipp.h>
#include <ipp/ipp_impl.h>
#include <ipp/ipgpc/ipgpc.h>

#include <sys/pattr.h>
#include <inet/dccp/dccp_ip.h>
#include <inet/dccp/dccp_impl.h>
#endif /* ! codereview */
#include <inet/ipclassifier.h>
#include <inet/sctp_ip.h>
#include <inet/sctp/sctp_impl.h>
#include <inet/udp_impl.h>
#include <inet/rawip_impl.h>
#include <inet/rts_impl.h>

#include <sys/tsol/label.h>
#include <sys/tsol/tnet.h>

#include <sys/squeue_impl.h>
```

```
#include <inet/ip_arp.h>

#include <sys/clock_impl.h>     /* For LBOLT_FASTPATH{,64} */

/*
 * Values for squeue switch:
 * IP_SQUEUE_ENTER_NODRAIN: SQ_NODRAIN
 * IP_SQUEUE_ENTER: SQ_PROCESS
 * IP_SQUEUE_FILL: SQ_FILL
 */
int ip_squeue_enter = IP_SQUEUE_ENTER;  /* Setable in /etc/system */

int ip_squeue_flag;

/*
 * Setable in /etc/system
 */
int ip_poll_normal_ms = 100;
int ip_poll_normal_ticks = 0;
int ip_modclose_ackwait_ms = 3000;

/*
 * It would be nice to have these present only in DEBUG systems, but the
 * current design of the global symbol checking logic requires them to be
 * unconditionally present.
 */
uint_t ip_thread_data;                  /* TSD key for debug support */
krwlock_t ip_thread_rwlock;
list_t  ip_thread_list;

/*
 * Structure to represent a linked list of msgblks. Used by ip_snmp_ functions.
 */

struct listptr_s {
        mblk_t  *lp_head;       /* pointer to the head of the list */
        mblk_t  *lp_tail;       /* pointer to the tail of the list */
};

typedef struct listptr_s listptr_t;

/*
 * This is used by ip_snmp_get_mib2_ip_route_media and
 * ip_snmp_get_mib2_ip6_route_media to carry the lists of return data.
 */
typedef struct iproutedata_s {
        uint_t          ird_idx;
        uint_t          ird_flags;      /* see below */
        listptr_t       ird_route;      /* ipRouteEntryTable */
        listptr_t       ird_netmedia;   /* ipNetToMediaEntryTable */
        listptr_t       ird_attrs;      /* ipRouteAttributeTable */
} iproutedata_t;

/* Include ire_testhidden and IRE_IF_CLONE routes */
#define IRD_REPORT_ALL  0x01

/*
 * Cluster specific hooks. These should be NULL when booted as a non-cluster
 */

/*
 * Hook functions to enable cluster networking
 * On non-clustered systems these vectors must always be NULL.
 *
 * Hook function to Check ip specified ip address is a shared ip address
 * in the cluster
```



```
 *
 */
int (*cl_inet_isclusterwide)(netstackid_t stack_id, uint8_t protocol,
    sa_family_t addr_family, uint8_t *laddrp, void *args) = NULL;

/*
 * Hook function to generate cluster wide ip fragment identifier
 */
uint32_t (*cl_inet_ipident)(netstackid_t stack_id, uint8_t protocol,
    sa_family_t addr_family, uint8_t *laddrp, uint8_t *faddrp,
    void *args) = NULL;

/*
 * Hook function to generate cluster wide SPI.
 */
void (*cl_inet_getspi)(netstackid_t, uint8_t, uint8_t *, size_t,
    void *) = NULL;

/*
 * Hook function to verify if the SPI is already utlized.
 */

int (*cl_inet_checkspi)(netstackid_t, uint8_t, uint32_t, void *) = NULL;

/*
 * Hook function to delete the SPI from the cluster wide repository.
 */

void (*cl_inet_deletespi)(netstackid_t, uint8_t, uint32_t, void *) = NULL;

/*
 * Hook function to inform the cluster when packet received on an IDLE SA
 */

void (*cl_inet_idlesa)(netstackid_t, uint8_t, uint32_t, sa_family_t,
    in6_addr_t, in6_addr_t, void *) = NULL;

/*
 * Synchronization notes:
 *
 * IP is a fully D_MP STREAMS module/driver. Thus it does not depend on any
 * MT level protection given by STREAMS. IP uses a combination of its own
 * internal serialization mechanism and standard Solaris locking techniques.
 * The internal serialization is per phyint.  This is used to serialize
 * plumbing operations, IPMP operations, most set ioctls, etc.
 *
 * Plumbing is a long sequence of operations involving message
 * exchanges between IP, ARP and device drivers. Many set ioctls are typically
 * involved in plumbing operations. A natural model is to serialize these
 * ioctls one per ill. For example plumbing of hme0 and qfe0 can go on in
 * parallel without any interference. But various set ioctls on hme0 are best
 * serialized, along with IPMP operations and processing of DLPI control
 * messages received from drivers on a per phyint basis. This serialization is
 * provided by the ipsq_t and primitives operating on this. Details can
 * be found in ip_if.c above the core primitives operating on ipsq_t.
 *
 * Lookups of an ipif or ill by a thread return a refheld ipif / ill.
 * Simiarly lookup of an ire by a thread also returns a refheld ire.
 * In addition ipif's and ill's referenced by the ire are also indirectly
 * refheld. Thus no ipif or ill can vanish as long as an ipif is refheld
 * directly or indirectly. For example an SIOCSLIFADDR ioctl that changes the
 * address of an ipif has to go through the ipsq_t. This ensures that only
 * one such exclusive operation proceeds at any time on the ipif. It then
 * waits for all refcnts
 * associated with this ipif to come down to zero. The address is changed
 * only after the ipif has been quiesced. Then the ipif is brought up again.
```

```
 * More details are described above the comment in ip_sioctl_flags.
 *
 * Packet processing is based mostly on IREs and are fully multi-threaded
 * using standard Solaris MT techniques.
 *
 * There are explicit locks in IP to handle:
 * - The ip_g_head list maintained by mi_open_link() and friends.
 *
 * - The reassembly data structures (one lock per hash bucket)
 *
 * - conn_lock is meant to protect conn_t fields. The fields actually
 *   protected by conn_lock are documented in the conn_t definition.
 *
 * - ire_lock to protect some of the fields of the ire, IRE tables
 *   (one lock per hash bucket). Refer to ip_ire.c for details.
 *
 * - ndp_g_lock and ncec_lock for protecting NCEs.
 *
 * - ill_lock protects fields of the ill and ipif. Details in ip.h
 *
 * - ill_g_lock: This is a global reader/writer lock. Protects the following
 *      * The AVL tree based global multi list of all ills.
 *      * The linked list of all ipifs of an ill
 *      * The <ipsq-xop> mapping
 *      * <ill-phyint> association
 *   Insertion/deletion of an ill in the system, insertion/deletion of an ipif
 *   into an ill, changing the <ipsq-xop> mapping of an ill, changing the
 *   <ill-phyint> assoc of an ill will all have to hold the ill_g_lock as
 *   writer for the actual duration of the insertion/deletion/change.
 *
 * - ill_lock:  This is a per ill mutex.
 *   It protects some members of the ill_t struct; see ip.h for details.
 *   It also protects the <ill-phyint> assoc.
 *   It also protects the list of ipifs hanging off the ill.
 *
 * - ipsq_lock: This is a per ipsq_t mutex lock.
 *   This protects some members of the ipsq_t struct; see ip.h for details.
 *   It also protects the <ipsq-ipxop> mapping
 *
 * - ipx_lock: This is a per ipxop_t mutex lock.
 *   This protects some members of the ipxop_t struct; see ip.h for details.
 *
 * - phyint_lock: This is a per phyint mutex lock. Protects just the
 *   phyint_flags
 *
 * - ip_addr_avail_lock: This is used to ensure the uniqueness of IP addresses.
 *   This lock is held in ipif_up_done and the ipif is marked IPIF_UP and the
 *   uniqueness check also done atomically.
 *
 * - ill_g_usesrc_lock: This readers/writer lock protects the usesrc
 *   group list linked by ill_usesrc_grp_next. It also protects the
 *   ill_usesrc_ifindex field. It is taken as a writer when a member of the
 *   group is being added or deleted.  This lock is taken as a reader when
 *   walking the list/group(eg: to get the number of members in a usesrc group).
 *   Note, it is only necessary to take this lock if the ill_usesrc_grp_next
 *   field is changing state i.e from NULL to non-NULL or vice-versa. For
 *   example, it is not necessary to take this lock in the initial portion
 *   of ip_sioctl_slifusesrc or at all in ip_sioctl_flags since these
 *   operations are executed exclusively and that ensures that the "usesrc
 *   group state" cannot change. The "usesrc group state" change can happen
 *   only in the latter part of ip_sioctl_slifusesrc and in ill_delete.
 *
 * Changing <ill-phyint>, <ipsq-xop> assocications:
 *
 * To change the <ill-phyint> association, the ill_g_lock must be held
 * as writer, and the ill_locks of both the v4 and v6 instance of the ill
```



```
 * must be held.
 *
 * To change the <ipsq-xop> association, the ill_g_lock must be held as
 * writer, the ipsq_lock must be held, and one must be writer on the ipsq.
 * This is only done when ills are added or removed from IPMP groups.
 *
 * To add or delete an ipif from the list of ipifs hanging off the ill,
 * ill_g_lock (writer) and ill_lock must be held and the thread must be
 * a writer on the associated ipsq.
 *
 * To add or delete an ill to the system, the ill_g_lock must be held as
 * writer and the thread must be a writer on the associated ipsq.
 *
 * To add or delete an ilm to an ill, the ill_lock must be held and the thread
 * must be a writer on the associated ipsq.
 *
 * Lock hierarchy
 *
 * Some lock hierarchy scenarios are listed below.
 *
 * ill_g_lock -> conn_lock -> ill_lock -> ipsq_lock -> ipx_lock
 * ill_g_lock -> ill_lock(s) -> phyint_lock
 * ill_g_lock -> ndp_g_lock -> ill_lock -> ncec_lock
 * ill_g_lock -> ip_addr_avail_lock
 * conn_lock -> irb_lock -> ill_lock -> ire_lock
 * ill_g_lock -> ip_g_nd_lock
 * ill_g_lock -> ips_ipmp_lock -> ill_lock -> nce_lock
 * ill_g_lock -> ndp_g_lock -> ill_lock -> ncec_lock -> nce_lock
 * arl_lock -> ill_lock
 * ips_ire_dep_lock -> irb_lock
 *
 * When more than 1 ill lock is needed to be held, all ill lock addresses
 * are sorted on address and locked starting from highest addressed lock
 * downward.
 *
 * Multicast scenarios
 * ips_ill_g_lock -> ill_mcast_lock
 * conn_ilg_lock -> ips_ill_g_lock -> ill_lock
 * ill_mcast_serializer -> ill_mcast_lock -> ips_ipmp_lock -> ill_lock
 * ill_mcast_serializer -> ill_mcast_lock -> connf_lock -> conn_lock
 * ill_mcast_serializer -> ill_mcast_lock -> conn_ilg_lock
 * ill_mcast_serializer -> ill_mcast_lock -> ips_igmp_timer_lock
 *
 * IPsec scenarios
 *
 * ipsa_lock -> ill_g_lock -> ill_lock
 * ill_g_usesrc_lock -> ill_g_lock -> ill_lock
 *
 * Trusted Solaris scenarios
 *
 * igsa_lock -> gcgrp_rwlock -> gcgrp_lock
 * igsa_lock -> gcdb_lock
 * gcgrp_rwlock -> ire_lock
 * gcgrp_rwlock -> gcdb_lock
 *
 * squeue(sq_lock), flow related (ft_lock, fe_lock) locking
 *
 * cpu_lock --> ill_lock --> sqset_lock --> sq_lock
 * sq_lock -> conn_lock -> QLOCK(q)
 * ill_lock -> ft_lock -> fe_lock
 *
 * Routing/forwarding table locking notes:
 *
 * Lock acquisition order: Radix tree lock, irb_lock.
 * Requirements:
 * i.  Walker must not hold any locks during the walker callback.
```

```
392  * ii  Walker must not see a truncated tree during the walk because of any node
393  *     deletion.
394  * iii Existing code assumes ire_bucket is valid if it is non-null and is used
395  *     in many places in the code to walk the irb list. Thus even if all the
396  *     ires in a bucket have been deleted, we still can't free the radix node
397  *     until the ires have actually been inactive'd (freed).
398  *
399  * Tree traversal - Need to hold the global tree lock in read mode.
400  * Before dropping the global tree lock, need to either increment the ire_refcnt
401  * to ensure that the radix node can't be deleted.
402  *
403  * Tree add - Need to hold the global tree lock in write mode to add a
404  * radix node. To prevent the node from being deleted, increment the
405  * irb_refcnt, after the node is added to the tree. The ire itself is
406  * added later while holding the irb_lock, but not the tree lock.
407  *
408  * Tree delete - Need to hold the global tree lock and irb_lock in write mode.
409  * All associated ires must be inactive (i.e. freed), and irb_refcnt
410  * must be zero.
411  *
412  * Walker - Increment irb_refcnt before calling the walker callback. Hold the
413  * global tree lock (read mode) for traversal.
414  *
415  * IRE dependencies - In some cases we hold ips_ire_dep_lock across ire_refrele
416  * hence we will acquire irb_lock while holding ips_ire_dep_lock.
417  *
418  * IPsec notes :
419  *
420  * IP interacts with the IPsec code (AH/ESP) by storing IPsec attributes
421  * in the ip_xmit_attr_t ip_recv_attr_t. For outbound datagrams, the
422  * ip_xmit_attr_t has the
423  * information used by the IPsec code for applying the right level of
424  * protection. The information initialized by IP in the ip_xmit_attr_t
425  * is determined by the per-socket policy or global policy in the system.
426  * For inbound datagrams, the ip_recv_attr_t
427  * starts out with nothing in it. It gets filled
428  * with the right information if it goes through the AH/ESP code, which
429  * happens if the incoming packet is secure. The information initialized
430  * by AH/ESP, is later used by IP (during fanouts to ULP) to see whether
431  * the policy requirements needed by per-socket policy or global policy
432  * is met or not.
433  *
434  * For fully connected sockets i.e dst, src [addr, port] is known,
435  * conn_policy_cached is set indicating that policy has been cached.
436  * conn_in_enforce_policy may or may not be set depending on whether
437  * there is a global policy match or per-socket policy match.
438  * Policy inheriting happpens in ip_policy_set once the destination is known.
439  * Once the right policy is set on the conn_t, policy cannot change for
440  * this socket. This makes life simpler for TCP (UDP ?) where
441  * re-transmissions go out with the same policy. For symmetry, policy
442  * is cached for fully connected UDP sockets also. Thus if policy is cached,
443  * it also implies that policy is latched i.e policy cannot change
444  * on these sockets. As we have the right policy on the conn, we don't
445  * have to lookup global policy for every outbound and inbound datagram
446  * and thus serving as an optimization. Note that a global policy change
447  * does not affect fully connected sockets if they have policy. If fully
448  * connected sockets did not have any policy associated with it, global
449  * policy change may affect them.
450  *
451  * IP Flow control notes:
452  * ----------------------
453  * Non-TCP streams are flow controlled by IP. The way this is accomplished
454  * differs when ILL_CAPAB_DLD_DIRECT is enabled for that IP instance. When
455  * ILL_DIRECT_CAPABLE(ill) is TRUE, IP can do direct function calls into
456  * GLDv3. Otherwise packets are sent down to lower layers using STREAMS
457  * functions.
```



```
458  *
459  * Per Tx ring udp flow control:
460  * This is applicable only when ILL_CAPAB_DLD_DIRECT capability is set in
461  * the ill (i.e. ILL_DIRECT_CAPABLE(ill) is true).
462  *
463  * The underlying link can expose multiple Tx rings to the GLDv3 mac layer.
464  * To achieve best performance, outgoing traffic need to be fanned out among
465  * these Tx ring. mac_tx() is called (via str_mdata_fastpath_put()) to send
466  * traffic out of the NIC and it takes a fanout hint. UDP connections pass
467  * the address of connp as fanout hint to mac_tx(). Under flow controlled
468  * condition, mac_tx() returns a non-NULL cookie (ip_mac_tx_cookie_t). This
469  * cookie points to a specific Tx ring that is blocked. The cookie is used to
470  * hash into an idl_tx_list[] entry in idl_tx_list[] array. Each idl_tx_list_t
471  * point to drain_lists (idl_t's). These drain list will store the blocked UDP
472  * connp's. The drain list is not a single list but a configurable number of
473  * lists.
474  *
475  * The diagram below shows idl_tx_list_t's and their drain_lists. ip_stack_t
476  * has an array of idl_tx_list_t. The size of the array is TX_FANOUT_SIZE
477  * which is equal to 128. This array in turn contains a pointer to idl_t[],
478  * the ip drain list. The idl_t[] array size is MIN(max_ncpus, 8). The drain
479  * list will point to the list of connp's that are flow controlled.
480  *
481  *                      ---------------   -------   -------   -------
482  *                   |->|drain_list[0]|-->|connp|-->|connp|-->|connp|-->
483  *                   |  ---------------   -------   -------   -------
484  *                   |  ---------------   -------   -------   -------
485  *                   |->|drain_list[1]|-->|connp|-->|connp|-->|connp|-->
486  * ----------------  |  ---------------   -------   -------   -------
487  * |idl_tx_list[0]|->|  ---------------   -------   -------   -------
488  * ----------------  |->|drain_list[2]|-->|connp|-->|connp|-->|connp|-->
489  *                   |  ---------------   -------   -------   -------
490  *                   .        .             .         .         .
491  *                   |  ---------------   -------   -------   -------
492  *                   |->|drain_list[n]|-->|connp|-->|connp|-->|connp|-->
493  *                      ---------------   -------   -------   -------
494  *                      ---------------   -------   -------   -------
495  *                   |->|drain_list[0]|-->|connp|-->|connp|-->|connp|-->
496  *                   |  ---------------   -------   -------   -------
497  *                   |  ---------------   -------   -------   -------
498  * ----------------  |->|drain_list[1]|-->|connp|-->|connp|-->|connp|-->
499  * |idl_tx_list[1]|->|  ---------------   -------   -------   -------
500  * ----------------  |        .             .         .         .
501  *                   |  ---------------   -------   -------   -------
502  *                   |->|drain_list[n]|-->|connp|-->|connp|-->|connp|-->
503  *                      ---------------   -------   -------   -------
504  *    .....
505  * ----------------
506  * |idl_tx_list[n]|-> ...
507  * ----------------
508  *
509  * When mac_tx() returns a cookie, the cookie is hashed into an index into
510  * ips_idl_tx_list[], and conn_drain_insert() is called with the idl_tx_list
511  * to insert the conn onto.  conn_drain_insert() asserts flow control for the
512  * sockets via su_txq_full() (non-STREAMS) or QFULL on conn_wq (STREAMS).
513  * Further, conn_blocked is set to indicate that the conn is blocked.
514  *
515  * GLDv3 calls ill_flow_enable() when flow control is relieved.  The cookie
516  * passed in the call to ill_flow_enable() identifies the blocked Tx ring and
517  * is again hashed to locate the appropriate idl_tx_list, which is then
518  * drained via conn_walk_drain().  conn_walk_drain() goes through each conn in
519  * the drain list and calls conn_drain_remove() to clear flow control (via
520  * calling su_txq_full() or clearing QFULL), and remove the conn from the
521  * drain list.
522  *
523  * Note that the drain list is not a single list but a (configurable) array of
```

```
 * lists (8 elements by default).  Synchronization between drain insertion and
 * flow control wakeup is handled by using idl_txl->txl_lock, and only
 * conn_drain_insert() and conn_drain_remove() manipulate the drain list.
 *
 * Flow control via STREAMS is used when ILL_DIRECT_CAPABLE() returns FALSE.
 * On the send side, if the packet cannot be sent down to the driver by IP
 * (canput() fails), ip_xmit() drops the packet and returns EWOULDBLOCK to the
 * caller, who may then invoke ixa_check_drain_insert() to insert the conn on
 * the 0'th drain list.  When ip_wsrv() runs on the ill_wq because flow
 * control has been relieved, the blocked conns in the 0'th drain list are
 * drained as in the non-STREAMS case.
 *
 * In both the STREAMS and non-STREAMS cases, the sockfs upcall to set QFULL
 * is done when the conn is inserted into the drain list (conn_drain_insert())
 * and cleared when the conn is removed from the it (conn_drain_remove()).
 *
 * IPQOS notes:
 *
 * IPQoS Policies are applied to packets using IPPF (IP Policy framework)
 * and IPQoS modules. IPPF includes hooks in IP at different control points
 * (callout positions) which direct packets to IPQoS modules for policy
 * processing. Policies, if present, are global.
 *
 * The callout positions are located in the following paths:
 *              o local_in (packets destined for this host)
 *              o local_out (packets orginating from this host )
 *              o fwd_in  (packets forwarded by this m/c - inbound)
 *              o fwd_out (packets forwarded by this m/c - outbound)
 * Hooks at these callout points can be enabled/disabled using the ndd variable
 * ip_policy_mask (a bit mask with the 4 LSB indicating the callout positions).
 * By default all the callout positions are enabled.
 *
 * Outbound (local_out)
 * Hooks are placed in ire_send_wire_v4 and ire_send_wire_v6.
 *
 * Inbound (local_in)
 * Hooks are placed in ip_fanout_v4 and ip_fanout_v6.
 *
 * Forwarding (in and out)
 * Hooks are placed in ire_recv_forward_v4/v6.
 *
 * IP Policy Framework processing (IPPF processing)
 * Policy processing for a packet is initiated by ip_process, which ascertains
 * that the classifier (ipgpc) is loaded and configured, failing which the
 * packet resumes normal processing in IP. If the clasifier is present, the
 * packet is acted upon by one or more IPQoS modules (action instances), per
 * filters configured in ipgpc and resumes normal IP processing thereafter.
 * An action instance can drop a packet in course of its processing.
 *
 * Zones notes:
 *
 * The partitioning rules for networking are as follows:
 * 1) Packets coming from a zone must have a source address belonging to that
 * zone.
 * 2) Packets coming from a zone can only be sent on a physical interface on
 * which the zone has an IP address.
 * 3) Between two zones on the same machine, packet delivery is only allowed if
 * there's a matching route for the destination and zone in the forwarding
 * table.
 * 4) The TCP and UDP port spaces are per-zone; that is, two processes in
 * different zones can bind to the same port with the wildcard address
 * (INADDR_ANY).
 *
 * The granularity of interface partitioning is at the logical interface level.
 * Therefore, every zone has its own IP addresses, and incoming packets can be
 * attributed to a zone unambiguously. A logical interface is placed into a zone
```



```
 * using the SIOCSLIFZONE ioctl; this sets the ipif_zoneid field in the ipif_t
 * structure. Rule (1) is implemented by modifying the source address selection
 * algorithm so that the list of eligible addresses is filtered based on the
 * sending process zone.
 *
 * The Internet Routing Entries (IREs) are either exclusive to a zone or shared
 * across all zones, depending on their type. Here is the break-up:
 *
 * IRE type                             Shared/exclusive
 * --------                             ----------------
 * IRE_BROADCAST                        Exclusive
 * IRE_DEFAULT (default routes)         Shared (*)
 * IRE_LOCAL                            Exclusive (x)
 * IRE_LOOPBACK                         Exclusive
 * IRE_PREFIX (net routes)              Shared (*)
 * IRE_IF_NORESOLVER (interface routes) Exclusive
 * IRE_IF_RESOLVER (interface routes)   Exclusive
 * IRE_IF_CLONE (interface routes)      Exclusive
 * IRE_HOST (host routes)               Shared (*)
 *
 * (*) A zone can only use a default or off-subnet route if the gateway is
 * directly reachable from the zone, that is, if the gateway's address matches
 * one of the zone's logical interfaces.
 *
 * (x) IRE_LOCAL are handled a bit differently.
 * When ip_restrict_interzone_loopback is set (the default),
 * ire_route_recursive restricts loopback using an IRE_LOCAL
 * between zone to the case when L2 would have conceptually looped the packet
 * back, i.e. the loopback which is required since neither Ethernet drivers
 * nor Ethernet hardware loops them back. This is the case when the normal
 * routes (ignoring IREs with different zoneids) would send out the packet on
 * the same ill as the ill with which is IRE_LOCAL is associated.
 *
 * Multiple zones can share a common broadcast address; typically all zones
 * share the 255.255.255.255 address. Incoming as well as locally originated
 * broadcast packets must be dispatched to all the zones on the broadcast
 * network. For directed broadcasts (e.g. 10.16.72.255) this is not trivial
 * since some zones may not be on the 10.16.72/24 network. To handle this, each
 * zone has its own set of IRE_BROADCAST entries; then, broadcast packets are
 * sent to every zone that has an IRE_BROADCAST entry for the destination
 * address on the input ill, see ip_input_broadcast().
 *
 * Applications in different zones can join the same multicast group address.
 * The same logic applies for multicast as for broadcast. ip_input_multicast
 * dispatches packets to all zones that have members on the physical interface.
 */

/*
 * Squeue Fanout flags:
 *      0: No fanout.
 *      1: Fanout across all squeues
 */
boolean_t       ip_squeue_fanout = 0;

/*
 * Maximum dups allowed per packet.
 */
uint_t ip_max_frag_dups = 10;

static int      ip_open(queue_t *q, dev_t *devp, int flag, int sflag,
                    cred_t *credp, boolean_t isv6);
static mblk_t   *ip_xmit_attach_llhdr(mblk_t *, nce_t *);

static boolean_t icmp_inbound_verify_v4(mblk_t *, icmph_t *, ip_recv_attr_t *);
static void     icmp_inbound_too_big_v4(icmph_t *, ip_recv_attr_t *);
static void     icmp_inbound_error_fanout_v4(mblk_t *, icmph_t *,
```

```
656     ip_recv_attr_t *);
657 static void     icmp_options_update(ipha_t *);
658 static void     icmp_param_problem(mblk_t *, uint8_t,  ip_recv_attr_t *);
659 static void     icmp_pkt(mblk_t *, void *, size_t, ip_recv_attr_t *);
660 static mblk_t   *icmp_pkt_err_ok(mblk_t *, ip_recv_attr_t *);
661 static void     icmp_redirect_v4(mblk_t *mp, ipha_t *, icmph_t *,
662     ip_recv_attr_t *);
663 static void     icmp_send_redirect(mblk_t *, ipaddr_t, ip_recv_attr_t *);
664 static void     icmp_send_reply_v4(mblk_t *, ipha_t *, icmph_t *,
665     ip_recv_attr_t *);

667 mblk_t          *ip_dlpi_alloc(size_t, t_uscalar_t);
668 char            *ip_dot_addr(ipaddr_t, char *);
669 mblk_t          *ip_carve_mp(mblk_t **, ssize_t);
670 int             ip_close(queue_t *, int);
671 static char     *ip_dot_saddr(uchar_t *, char *);
672 static void     ip_lrput(queue_t *, mblk_t *);
673 ipaddr_t        ip_net_mask(ipaddr_t);
674 char            *ip_nv_lookup(nv_t *, int);
675 void    ip_rput(queue_t *, mblk_t *);
676 static void     ip_rput_dlpi_writer(ipsq_t *dummy_sq, queue_t *q, mblk_t *mp,
677                     void *dummy_arg);
678 int             ip_snmp_get(queue_t *, mblk_t *, int, boolean_t);
679 static mblk_t   *ip_snmp_get_mib2_ip(queue_t *, mblk_t *,
680                     mib2_ipIfStatsEntry_t *, ip_stack_t *, boolean_t);
681 static mblk_t   *ip_snmp_get_mib2_ip_traffic_stats(queue_t *, mblk_t *,
682                     ip_stack_t *, boolean_t);
683 static mblk_t   *ip_snmp_get_mib2_ip6(queue_t *, mblk_t *, ip_stack_t *,
684                     boolean_t);
685 static mblk_t   *ip_snmp_get_mib2_icmp(queue_t *, mblk_t *, ip_stack_t *ipst);
686 static mblk_t   *ip_snmp_get_mib2_icmp6(queue_t *, mblk_t *, ip_stack_t *ipst);
687 static mblk_t   *ip_snmp_get_mib2_igmp(queue_t *, mblk_t *, ip_stack_t *ipst);
688 static mblk_t   *ip_snmp_get_mib2_multi(queue_t *, mblk_t *, ip_stack_t *ipst);
689 static mblk_t   *ip_snmp_get_mib2_ip_addr(queue_t *, mblk_t *,
690                     ip_stack_t *ipst, boolean_t);
691 static mblk_t   *ip_snmp_get_mib2_ip6_addr(queue_t *, mblk_t *,
692                     ip_stack_t *ipst, boolean_t);
693 static mblk_t   *ip_snmp_get_mib2_ip_group_src(queue_t *, mblk_t *,
694                     ip_stack_t *ipst);
695 static mblk_t   *ip_snmp_get_mib2_ip6_group_src(queue_t *, mblk_t *,
696                     ip_stack_t *ipst);
697 static mblk_t   *ip_snmp_get_mib2_ip_group_mem(queue_t *, mblk_t *,
698                     ip_stack_t *ipst);
699 static mblk_t   *ip_snmp_get_mib2_ip6_group_mem(queue_t *, mblk_t *,
700                     ip_stack_t *ipst);
701 static mblk_t   *ip_snmp_get_mib2_virt_multi(queue_t *, mblk_t *,
702                     ip_stack_t *ipst);
703 static mblk_t   *ip_snmp_get_mib2_multi_rtable(queue_t *, mblk_t *,
704                     ip_stack_t *ipst);
705 static mblk_t   *ip_snmp_get_mib2_ip_route_media(queue_t *, mblk_t *, int,
706                     ip_stack_t *ipst);
707 static mblk_t   *ip_snmp_get_mib2_ip6_route_media(queue_t *, mblk_t *, int,
708                     ip_stack_t *ipst);
709 static void     ip_snmp_get2_v4(ire_t *, iproutedata_t *);
710 static void     ip_snmp_get2_v6_route(ire_t *, iproutedata_t *);
711 static int      ip_snmp_get2_v4_media(ncec_t *, iproutedata_t *);
712 static int      ip_snmp_get2_v6_media(ncec_t *, iproutedata_t *);
713 int             ip_snmp_set(queue_t *, int, int, uchar_t *, int);

715 static mblk_t   *ip_fragment_copyhdr(uchar_t *, int, int, ip_stack_t *,
716                     mblk_t *);

718 static void     conn_drain_init(ip_stack_t *);
719 static void     conn_drain_fini(ip_stack_t *);
720 static void     conn_drain(conn_t *connp, boolean_t closing);
```



```
722 static void     conn_walk_drain(ip_stack_t *, idl_tx_list_t *);
723 static void     conn_walk_sctp(pfv_t, void *, zoneid_t, netstack_t *);

725 static void     *ip_stack_init(netstackid_t stackid, netstack_t *ns);
726 static void     ip_stack_shutdown(netstackid_t stackid, void *arg);
727 static void     ip_stack_fini(netstackid_t stackid, void *arg);

729 static int      ip_multirt_apply_membership(int (*fn)(conn_t *, boolean_t,
730     const in6_addr_t *, ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *),
731     ire_t *, conn_t *, boolean_t, const in6_addr_t *,  mcast_record_t,
732     const in6_addr_t *);

734 static int      ip_squeue_switch(int);

736 static void     *ip_kstat_init(netstackid_t, ip_stack_t *);
737 static void     ip_kstat_fini(netstackid_t, kstat_t *);
738 static int      ip_kstat_update(kstat_t *kp, int rw);
739 static void     *icmp_kstat_init(netstackid_t);
740 static void     icmp_kstat_fini(netstackid_t, kstat_t *);
741 static int      icmp_kstat_update(kstat_t *kp, int rw);
742 static void     *ip_kstat2_init(netstackid_t, ip_stat_t *);
743 static void     ip_kstat2_fini(netstackid_t, kstat_t *);

745 static void     ipobs_init(ip_stack_t *);
746 static void     ipobs_fini(ip_stack_t *);

748 static int      ip_tp_cpu_update(cpu_setup_t, int, void *);

750 ipaddr_t        ip_g_all_ones = IP_HOST_MASK;

752 static long ip_rput_pullups;
753 int     dohwcksum = 1;  /* use h/w cksum if supported by the hardware */

755 vmem_t *ip_minor_arena_sa; /* for minor nos. from INET_MIN_DEV+2 thru 2^^18-1 */
756 vmem_t *ip_minor_arena_la; /* for minor nos. from 2^^18 thru 2^^32-1 */

758 int     ip_debug;

760 /*
761  * Multirouting/CGTP stuff
762  */
763 int     ip_cgtp_filter_rev = CGTP_FILTER_REV;   /* CGTP hooks version */

765 /*
766  * IP tunables related declarations. Definitions are in ip_tunables.c
767  */
768 extern mod_prop_info_t ip_propinfo_tbl[];
769 extern int ip_propinfo_count;

771 /*
772  * Table of IP ioctls encoding the various properties of the ioctl and
773  * indexed based on the last byte of the ioctl command. Occasionally there
774  * is a clash, and there is more than 1 ioctl with the same last byte.
775  * In such a case 1 ioctl is encoded in the ndx table and the remaining
776  * ioctls are encoded in the misc table. An entry in the ndx table is
777  * retrieved by indexing on the last byte of the ioctl command and comparing
778  * the ioctl command with the value in the ndx table. In the event of a
779  * mismatch the misc table is then searched sequentially for the desired
780  * ioctl command.
781  *
782  * Entry: <command> <copyin_size> <flags> <cmd_type> <function> <restart_func>
783  */
784 ip_ioctl_cmd_t ip_ndx_ioctl_table[] = {
785         /* 000 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
786         /* 001 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
787         /* 002 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
```

```
788         /* 003 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
789         /* 004 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
790         /* 005 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
791         /* 006 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
792         /* 007 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
793         /* 008 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
794         /* 009 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

796         /* 010 */ { SIOCADDRT,  sizeof (struct rtentry), IPI_PRIV,
797                         MISC_CMD, ip_siocaddrt, NULL },
798         /* 011 */ { SIOCDELRT,  sizeof (struct rtentry), IPI_PRIV,
799                         MISC_CMD, ip_siocdelrt, NULL },

801         /* 012 */ { SIOCSIFADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
802                         IF_CMD, ip_sioctl_addr, ip_sioctl_addr_restart },
803         /* 013 */ { SIOCGIFADDR, sizeof (struct ifreq), IPI_GET_CMD,
804                         IF_CMD, ip_sioctl_get_addr, NULL },

806         /* 014 */ { SIOCSIFDSTADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
807                         IF_CMD, ip_sioctl_dstaddr, ip_sioctl_dstaddr_restart },
808         /* 015 */ { SIOCGIFDSTADDR, sizeof (struct ifreq),
809                         IPI_GET_CMD, IF_CMD, ip_sioctl_get_dstaddr, NULL },

811         /* 016 */ { SIOCSIFFLAGS, sizeof (struct ifreq),
812                         IPI_PRIV | IPI_WR,
813                         IF_CMD, ip_sioctl_flags, ip_sioctl_flags_restart },
814         /* 017 */ { SIOCGIFFLAGS, sizeof (struct ifreq),
815                         IPI_MODOK | IPI_GET_CMD,
816                         IF_CMD, ip_sioctl_get_flags, NULL },

818         /* 018 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
819         /* 019 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

821         /* copyin size cannot be coded for SIOCGIFCONF */
822         /* 020 */ { O_SIOCGIFCONF, 0, IPI_GET_CMD,
823                         MISC_CMD, ip_sioctl_get_ifconf, NULL },

825         /* 021 */ { SIOCSIFMTU, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
826                         IF_CMD, ip_sioctl_mtu, NULL },
827         /* 022 */ { SIOCGIFMTU, sizeof (struct ifreq), IPI_GET_CMD,
828                         IF_CMD, ip_sioctl_get_mtu, NULL },
829         /* 023 */ { SIOCGIFBRDADDR, sizeof (struct ifreq),
830                         IPI_GET_CMD, IF_CMD, ip_sioctl_get_brdaddr, NULL },
831         /* 024 */ { SIOCSIFBRDADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
832                         IF_CMD, ip_sioctl_brdaddr, NULL },
833         /* 025 */ { SIOCGIFNETMASK, sizeof (struct ifreq),
834                         IPI_GET_CMD, IF_CMD, ip_sioctl_get_netmask, NULL },
835         /* 026 */ { SIOCSIFNETMASK, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
836                         IF_CMD, ip_sioctl_netmask, ip_sioctl_netmask_restart },
837         /* 027 */ { SIOCGIFMETRIC, sizeof (struct ifreq),
838                         IPI_GET_CMD, IF_CMD, ip_sioctl_get_metric, NULL },
839         /* 028 */ { SIOCSIFMETRIC, sizeof (struct ifreq), IPI_PRIV,
840                         IF_CMD, ip_sioctl_metric, NULL },
841         /* 029 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

843         /* See 166-168 below for extended SIOC*XARP ioctls */
844         /* 030 */ { SIOCSARP, sizeof (struct arpreq), IPI_PRIV | IPI_WR,
845                         ARP_CMD, ip_sioctl_arp, NULL },
846         /* 031 */ { SIOCGARP, sizeof (struct arpreq), IPI_GET_CMD,
847                         ARP_CMD, ip_sioctl_arp, NULL },
848         /* 032 */ { SIOCDARP, sizeof (struct arpreq), IPI_PRIV | IPI_WR,
849                         ARP_CMD, ip_sioctl_arp, NULL },

851         /* 033 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
852         /* 034 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
853         /* 035 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
```



```
854         /* 036 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
855         /* 037 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
856         /* 038 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
857         /* 039 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
858         /* 040 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
859         /* 041 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
860         /* 042 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
861         /* 043 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
862         /* 044 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
863         /* 045 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
864         /* 046 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
865         /* 047 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
866         /* 048 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
867         /* 049 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
868         /* 050 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
869         /* 051 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
870         /* 052 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
871         /* 053 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

873         /* 054 */ { IF_UNITSEL, sizeof (int), IPI_PRIV | IPI_WR | IPI_MODOK,
874                         MISC_CMD, if_unitsel, if_unitsel_restart },

876         /* 055 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
877         /* 056 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
878         /* 057 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
879         /* 058 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
880         /* 059 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
881         /* 060 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
882         /* 061 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
883         /* 062 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
884         /* 063 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
885         /* 064 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
886         /* 065 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
887         /* 066 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
888         /* 067 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
889         /* 068 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
890         /* 069 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
891         /* 070 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
892         /* 071 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
893         /* 072 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

895         /* 073 */ { SIOCSIFNAME, sizeof (struct ifreq),
896                         IPI_PRIV | IPI_WR | IPI_MODOK,
897                         IF_CMD, ip_sioctl_sifname, NULL },

899         /* 074 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
900         /* 075 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
901         /* 076 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
902         /* 077 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
903         /* 078 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
904         /* 079 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
905         /* 080 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
906         /* 081 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
907         /* 082 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
908         /* 083 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
909         /* 084 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
910         /* 085 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
911         /* 086 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

913         /* 087 */ { SIOCGIFNUM, sizeof (int), IPI_GET_CMD,
914                         MISC_CMD, ip_sioctl_get_ifnum, NULL },
915         /* 088 */ { SIOCGIFMUXID, sizeof (struct ifreq), IPI_GET_CMD,
916                         IF_CMD, ip_sioctl_get_muxid, NULL },
917         /* 089 */ { SIOCSIFMUXID, sizeof (struct ifreq),
918                         IPI_PRIV | IPI_WR, IF_CMD, ip_sioctl_muxid, NULL },
```

```
920         /* Both if and lif variants share same func */
921         /* 090 */ { SIOCGIFINDEX, sizeof (struct ifreq), IPI_GET_CMD,
922                         IF_CMD, ip_sioctl_get_lifindex, NULL },
923         /* Both if and lif variants share same func */
924         /* 091 */ { SIOCSIFINDEX, sizeof (struct ifreq),
925                         IPI_PRIV | IPI_WR, IF_CMD, ip_sioctl_slifindex, NULL },

927         /* copyin size cannot be coded for SIOCGIFCONF */
928         /* 092 */ { SIOCGIFCONF, 0, IPI_GET_CMD,
929                         MISC_CMD, ip_sioctl_get_ifconf, NULL },
930         /* 093 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
931         /* 094 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
932         /* 095 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
933         /* 096 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
934         /* 097 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
935         /* 098 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
936         /* 099 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
937         /* 100 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
938         /* 101 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
939         /* 102 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
940         /* 103 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
941         /* 104 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
942         /* 105 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
943         /* 106 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
944         /* 107 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
945         /* 108 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
946         /* 109 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

948         /* 110 */ { SIOCLIFREMOVEIF, sizeof (struct lifreq),
949                         IPI_PRIV | IPI_WR, LIF_CMD, ip_sioctl_removeif,
950                         ip_sioctl_removeif_restart },
951         /* 111 */ { SIOCLIFADDIF, sizeof (struct lifreq),
952                         IPI_GET_CMD | IPI_PRIV | IPI_WR,
953                         LIF_CMD, ip_sioctl_addif, NULL },
954 #define SIOCLIFADDR_NDX 112
955         /* 112 */ { SIOCSLIFADDR, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
956                         LIF_CMD, ip_sioctl_addr, ip_sioctl_addr_restart },
957         /* 113 */ { SIOCGLIFADDR, sizeof (struct lifreq),
958                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_addr, NULL },
959         /* 114 */ { SIOCSLIFDSTADDR, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
960                         LIF_CMD, ip_sioctl_dstaddr, ip_sioctl_dstaddr_restart },
961         /* 115 */ { SIOCGLIFDSTADDR, sizeof (struct lifreq),
962                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_dstaddr, NULL },
963         /* 116 */ { SIOCSLIFFLAGS, sizeof (struct lifreq),
964                         IPI_PRIV | IPI_WR,
965                         LIF_CMD, ip_sioctl_flags, ip_sioctl_flags_restart },
966         /* 117 */ { SIOCGLIFFLAGS, sizeof (struct lifreq),
967                         IPI_GET_CMD | IPI_MODOK,
968                         LIF_CMD, ip_sioctl_get_flags, NULL },

970         /* 118 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
971         /* 119 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

973         /* 120 */ { O_SIOCGLIFCONF, 0, IPI_GET_CMD, MISC_CMD,
974                         ip_sioctl_get_lifconf, NULL },
975         /* 121 */ { SIOCSLIFMTU, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
976                         LIF_CMD, ip_sioctl_mtu, NULL },
977         /* 122 */ { SIOCGLIFMTU, sizeof (struct lifreq), IPI_GET_CMD,
978                         LIF_CMD, ip_sioctl_get_mtu, NULL },
979         /* 123 */ { SIOCGLIFBRDADDR, sizeof (struct lifreq),
980                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_brdaddr, NULL },
981         /* 124 */ { SIOCSLIFBRDADDR, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
982                         LIF_CMD, ip_sioctl_brdaddr, NULL },
983         /* 125 */ { SIOCGLIFNETMASK, sizeof (struct lifreq),
984                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_netmask, NULL },
985         /* 126 */ { SIOCSLIFNETMASK, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
```



```
986                         LIF_CMD, ip_sioctl_netmask, ip_sioctl_netmask_restart },
987         /* 127 */ { SIOCGLIFMETRIC, sizeof (struct lifreq),
988                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_metric, NULL },
989         /* 128 */ { SIOCSLIFMETRIC, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
990                         LIF_CMD, ip_sioctl_metric, NULL },
991         /* 129 */ { SIOCSLIFNAME, sizeof (struct lifreq),
992                         IPI_PRIV | IPI_WR | IPI_MODOK,
993                         LIF_CMD, ip_sioctl_slifname,
994                         ip_sioctl_slifname_restart },

996         /* 130 */ { SIOCGLIFNUM, sizeof (struct lifnum), IPI_GET_CMD,
997                         MISC_CMD, ip_sioctl_get_lifnum, NULL },
998         /* 131 */ { SIOCGLIFMUXID, sizeof (struct lifreq),
999                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_muxid, NULL },
1000        /* 132 */ { SIOCSLIFMUXID, sizeof (struct lifreq),
1001                        IPI_PRIV | IPI_WR, LIF_CMD, ip_sioctl_muxid, NULL },
1002        /* 133 */ { SIOCGLIFINDEX, sizeof (struct lifreq),
1003                        IPI_GET_CMD, LIF_CMD, ip_sioctl_get_lifindex, 0 },
1004        /* 134 */ { SIOCSLIFINDEX, sizeof (struct lifreq),
1005                        IPI_PRIV | IPI_WR, LIF_CMD, ip_sioctl_slifindex, 0 },
1006        /* 135 */ { SIOCSLIFTOKEN, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
1007                        LIF_CMD, ip_sioctl_token, NULL },
1008        /* 136 */ { SIOCGLIFTOKEN, sizeof (struct lifreq),
1009                        IPI_GET_CMD, LIF_CMD, ip_sioctl_get_token, NULL },
1010        /* 137 */ { SIOCSLIFSUBNET, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
1011                        LIF_CMD, ip_sioctl_subnet, ip_sioctl_subnet_restart },
1012        /* 138 */ { SIOCGLIFSUBNET, sizeof (struct lifreq),
1013                        IPI_GET_CMD, LIF_CMD, ip_sioctl_get_subnet, NULL },
1014        /* 139 */ { SIOCSLIFLNKINFO, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
1015                        LIF_CMD, ip_sioctl_lnkinfo, NULL },

1017        /* 140 */ { SIOCGLIFLNKINFO, sizeof (struct lifreq),
1018                        IPI_GET_CMD, LIF_CMD, ip_sioctl_get_lnkinfo, NULL },
1019        /* 141 */ { SIOCLIFDELND, sizeof (struct lifreq), IPI_PRIV,
1020                        LIF_CMD, ip_siocdelndp_v6, NULL },
1021        /* 142 */ { SIOCLIFGETND, sizeof (struct lifreq), IPI_GET_CMD,
1022                        LIF_CMD, ip_siocqueryndp_v6, NULL },
1023        /* 143 */ { SIOCLIFSETND, sizeof (struct lifreq), IPI_PRIV,
1024                        LIF_CMD, ip_siocsetndp_v6, NULL },
1025        /* 144 */ { SIOCTMYADDR, sizeof (struct sioc_addrreq), IPI_GET_CMD,
1026                        MISC_CMD, ip_sioctl_tmyaddr, NULL },
1027        /* 145 */ { SIOCTONLINK, sizeof (struct sioc_addrreq), IPI_GET_CMD,
1028                        MISC_CMD, ip_sioctl_tonlink, NULL },
1029        /* 146 */ { SIOCTMYSITE, sizeof (struct sioc_addrreq), 0,
1030                        MISC_CMD, ip_sioctl_tmysite, NULL },
1031        /* 147 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1032        /* 148 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1033        /* IPSECioctls handled in ip_sioctl_copyin_setup itself */
1034        /* 149 */ { SIOCFIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },
1035        /* 150 */ { SIOCSIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },
1036        /* 151 */ { SIOCDIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },
1037        /* 152 */ { SIOCLIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },

1039        /* 153 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

1041        /* 154 */ { SIOCGLIFBINDING, sizeof (struct lifreq), IPI_GET_CMD,
1042                        LIF_CMD, ip_sioctl_get_binding, NULL },
1043        /* 155 */ { SIOCSLIFGROUPNAME, sizeof (struct lifreq),
1044                        IPI_PRIV | IPI_WR,
1045                        LIF_CMD, ip_sioctl_groupname, ip_sioctl_groupname },
1046        /* 156 */ { SIOCGLIFGROUPNAME, sizeof (struct lifreq),
1047                        IPI_GET_CMD, LIF_CMD, ip_sioctl_get_groupname, NULL },
1048        /* 157 */ { SIOCGLIFGROUPINFO, sizeof (lifgroupinfo_t),
1049                        IPI_GET_CMD, MISC_CMD, ip_sioctl_groupinfo, NULL },

1051        /* Leave 158-160 unused; used to be SIOC*IFARP ioctls */
```

```
1052         /* 158 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1053         /* 159 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1054         /* 160 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

1056         /* 161 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

1058         /* These are handled in ip_sioctl_copyin_setup itself */
1059         /* 162 */ { SIOCGIP6ADDRPOLICY, 0, IPI_NULL_BCONT,
1060                         MISC_CMD, NULL, NULL },
1061         /* 163 */ { SIOCSIP6ADDRPOLICY, 0, IPI_PRIV | IPI_NULL_BCONT,
1062                         MISC_CMD, NULL, NULL },
1063         /* 164 */ { SIOCGDSTINFO, 0, IPI_GET_CMD, MISC_CMD, NULL, NULL },

1065         /* 165 */ { SIOCGLIFCONF, 0, IPI_GET_CMD, MISC_CMD,
1066                         ip_sioctl_get_lifconf, NULL },

1068         /* 166 */ { SIOCSXARP, sizeof (struct xarpreq), IPI_PRIV | IPI_WR,
1069                         XARP_CMD, ip_sioctl_arp, NULL },
1070         /* 167 */ { SIOCGXARP, sizeof (struct xarpreq), IPI_GET_CMD,
1071                         XARP_CMD, ip_sioctl_arp, NULL },
1072         /* 168 */ { SIOCDXARP, sizeof (struct xarpreq), IPI_PRIV | IPI_WR,
1073                         XARP_CMD, ip_sioctl_arp, NULL },

1075         /* SIOCPOPSOCKFS is not handled by IP */
1076         /* 169 */ { IPI_DONTCARE /* SIOCPOPSOCKFS */, 0, 0, 0, NULL, NULL },

1078         /* 170 */ { SIOCGLIFZONE, sizeof (struct lifreq),
1079                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_lifzone, NULL },
1080         /* 171 */ { SIOCSLIFZONE, sizeof (struct lifreq),
1081                         IPI_PRIV | IPI_WR, LIF_CMD, ip_sioctl_slifzone,
1082                         ip_sioctl_slifzone_restart },
1083         /* 172-174 are SCTP ioctls and not handled by IP */
1084         /* 172 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1085         /* 173 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1086         /* 174 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1087         /* 175 */ { SIOCGLIFUSESRC, sizeof (struct lifreq),
1088                         IPI_GET_CMD, LIF_CMD,
1089                         ip_sioctl_get_lifusesrc, 0 },
1090         /* 176 */ { SIOCSLIFUSESRC, sizeof (struct lifreq),
1091                         IPI_PRIV | IPI_WR,
1092                         LIF_CMD, ip_sioctl_slifusesrc,
1093                         NULL },
1094         /* 177 */ { SIOCGLIFSRCOF, 0, IPI_GET_CMD, MISC_CMD,
1095                         ip_sioctl_get_lifsrcof, NULL },
1096         /* 178 */ { SIOCGMSFILTER, sizeof (struct group_filter), IPI_GET_CMD,
1097                         MSFILT_CMD, ip_sioctl_msfilter, NULL },
1098         /* 179 */ { SIOCSMSFILTER, sizeof (struct group_filter), 0,
1099                         MSFILT_CMD, ip_sioctl_msfilter, NULL },
1100         /* 180 */ { SIOCGIPMSFILTER, sizeof (struct ip_msfilter), IPI_GET_CMD,
1101                         MSFILT_CMD, ip_sioctl_msfilter, NULL },
1102         /* 181 */ { SIOCSIPMSFILTER, sizeof (struct ip_msfilter), 0,
1103                         MSFILT_CMD, ip_sioctl_msfilter, NULL },
1104         /* 182 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1105         /* SIOCSENABLESDP is handled by SDP */
1106         /* 183 */ { IPI_DONTCARE /* SIOCSENABLESDP */, 0, 0, 0, NULL, NULL },
1107         /* 184 */ { IPI_DONTCARE /* SIOCSQPTR */, 0, 0, 0, NULL, NULL },
1108         /* 185 */ { SIOCGIFHWADDR, sizeof (struct ifreq), IPI_GET_CMD,
1109                         IF_CMD, ip_sioctl_get_ifhwaddr, NULL },
1110         /* 186 */ { IPI_DONTCARE /* SIOCGSTAMP */, 0, 0, 0, NULL, NULL },
1111         /* 187 */ { SIOCILB, 0, IPI_PRIV | IPI_GET_CMD, MISC_CMD,
1112                         ip_sioctl_ilb_cmd, NULL },
1113         /* 188 */ { SIOCGETPROP, 0, IPI_GET_CMD, 0, NULL, NULL },
1114         /* 189 */ { SIOCSETPROP, 0, IPI_PRIV | IPI_WR, 0, NULL, NULL},
1115         /* 190 */ { SIOCGLIFDADSTATE, sizeof (struct lifreq),
1116                         IPI_GET_CMD, LIF_CMD, ip_sioctl_get_dadstate, NULL },
1117         /* 191 */ { SIOCSLIFPREFIX, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
```



```
1118                         LIF_CMD, ip_sioctl_prefix, ip_sioctl_prefix_restart },
1119         /* 192 */ { SIOCGLIFHWADDR, sizeof (struct lifreq), IPI_GET_CMD,
1120                         LIF_CMD, ip_sioctl_get_lifhwaddr, NULL }
1121 };

1123 int ip_ndx_ioctl_count = sizeof (ip_ndx_ioctl_table) / sizeof (ip_ioctl_cmd_t);

1125 ip_ioctl_cmd_t ip_misc_ioctl_table[] = {
1126         { I_LINK,       0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1127         { I_UNLINK,     0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1128         { I_PLINK,      0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1129         { I_PUNLINK,    0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1130         { ND_GET,       0, 0, 0, NULL, NULL },
1131         { ND_SET,       0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1132         { IP_IOCTL,     0, 0, 0, NULL, NULL },
1133         { SIOCGETVIFCNT, sizeof (struct sioc_vif_req), IPI_GET_CMD,
1134                 MISC_CMD, mrt_ioctl},
1135         { SIOCGETSGCNT, sizeof (struct sioc_sg_req), IPI_GET_CMD,
1136                 MISC_CMD, mrt_ioctl},
1137         { SIOCGETLSGCNT, sizeof (struct sioc_lsg_req), IPI_GET_CMD,
1138                 MISC_CMD, mrt_ioctl}
1139 };

1141 int ip_misc_ioctl_count =
1142     sizeof (ip_misc_ioctl_table) / sizeof (ip_ioctl_cmd_t);

1144 int     conn_drain_nthreads;            /* Number of drainers reqd. */
1145                                         /* Settable in /etc/system */
1146 /* Defined in ip_ire.c */
1147 extern uint32_t ip_ire_max_bucket_cnt, ip6_ire_max_bucket_cnt;
1148 extern uint32_t ip_ire_min_bucket_cnt, ip6_ire_min_bucket_cnt;
1149 extern uint32_t ip_ire_mem_ratio, ip_ire_cpu_ratio;

1151 static nv_t     ire_nv_arr[] = {
1152         { IRE_BROADCAST, "BROADCAST" },
1153         { IRE_LOCAL, "LOCAL" },
1154         { IRE_LOOPBACK, "LOOPBACK" },
1155         { IRE_DEFAULT, "DEFAULT" },
1156         { IRE_PREFIX, "PREFIX" },
1157         { IRE_IF_NORESOLVER, "IF_NORESOL" },
1158         { IRE_IF_RESOLVER, "IF_RESOLV" },
1159         { IRE_IF_CLONE, "IF_CLONE" },
1160         { IRE_HOST, "HOST" },
1161         { IRE_MULTICAST, "MULTICAST" },
1162         { IRE_NOROUTE, "NOROUTE" },
1163         { 0 }
1164 };

1166 nv_t    *ire_nv_tbl = ire_nv_arr;

1168 /* Simple ICMP IP Header Template */
1169 static ipha_t icmp_ipha = {
1170         IP_SIMPLE_HDR_VERSION, 0, 0, 0, 0, 0, IPPROTO_ICMP
1171 };

1173 struct module_info ip_mod_info = {
1174         IP_MOD_ID, IP_MOD_NAME, IP_MOD_MINPSZ, IP_MOD_MAXPSZ, IP_MOD_HIWAT,
1175         IP_MOD_LOWAT
1176 };

1178 /*
1179  * Duplicate static symbols within a module confuses mdb; so we avoid the
1180  * problem by making the symbols here distinct from those in udp.c.
1181  */

1183 /*
```

```
 * Entry points for IP as a device and as a module.
 * We have separate open functions for the /dev/ip and /dev/ip6 devices.
 */
static struct qinit iprinitv4 = {
        (pfi_t)ip_rput, NULL, ip_openv4, ip_close, NULL,
        &ip_mod_info
};

struct qinit iprinitv6 = {
        (pfi_t)ip_rput_v6, NULL, ip_openv6, ip_close, NULL,
        &ip_mod_info
};

static struct qinit ipwinit = {
        (pfi_t)ip_wput_nondata, (pfi_t)ip_wsrv, NULL, NULL, NULL,
        &ip_mod_info
};

static struct qinit iplrinit = {
        (pfi_t)ip_lrput, NULL, ip_openv4, ip_close, NULL,
        &ip_mod_info
};

static struct qinit iplwinit = {
        (pfi_t)ip_lwput, NULL, NULL, NULL, NULL,
        &ip_mod_info
};

/* For AF_INET aka /dev/ip */
struct streamtab ipinfov4 = {
        &iprinitv4, &ipwinit, &iplrinit, &iplwinit
};

/* For AF_INET6 aka /dev/ip6 */
struct streamtab ipinfov6 = {
        &iprinitv6, &ipwinit, &iplrinit, &iplwinit
};

#ifdef  DEBUG
boolean_t skip_sctp_cksum = B_FALSE;
#endif

/*
 * Generate an ICMP fragmentation needed message.
 * When called from ip_output side a minimal ip_recv_attr_t needs to be
 * constructed by the caller.
 */
void
icmp_frag_needed(mblk_t *mp, int mtu, ip_recv_attr_t *ira)
{
        icmph_t icmph;
        ip_stack_t      *ipst = ira->ira_ill->ill_ipst;

        mp = icmp_pkt_err_ok(mp, ira);
        if (mp == NULL)
                return;

        bzero(&icmph, sizeof (icmph_t));
        icmph.icmph_type = ICMP_DEST_UNREACHABLE;
        icmph.icmph_code = ICMP_FRAGMENTATION_NEEDED;
        icmph.icmph_du_mtu = htons((uint16_t)mtu);
        BUMP_MIB(&ipst->ips_icmp_mib, icmpOutFragNeeded);
        BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDestUnreachs);

        icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
}
```



```
/*
 * icmp_inbound_v4 deals with ICMP messages that are handled by IP.
 * If the ICMP message is consumed by IP, i.e., it should not be delivered
 * to any IPPROTO_ICMP raw sockets, then it returns NULL.
 * Likewise, if the ICMP error is misformed (too short, etc), then it
 * returns NULL. The caller uses this to determine whether or not to send
 * to raw sockets.
 *
 * All error messages are passed to the matching transport stream.
 *
 * The following cases are handled by icmp_inbound:
 * 1) It needs to send a reply back and possibly delivering it
 *    to the "interested" upper clients.
 * 2) Return the mblk so that the caller can pass it to the RAW socket clients.
 * 3) It needs to change some values in IP only.
 * 4) It needs to change some values in IP and upper layers e.g TCP
 *    by delivering an error to the upper layers.
 *
 * We handle the above three cases in the context of IPsec in the
 * following way :
 *
 * 1) Send the reply back in the same way as the request came in.
 *    If it came in encrypted, it goes out encrypted. If it came in
 *    clear, it goes out in clear. Thus, this will prevent chosen
 *    plain text attack.
 * 2) The client may or may not expect things to come in secure.
 *    If it comes in secure, the policy constraints are checked
 *    before delivering it to the upper layers. If it comes in
 *    clear, ipsec_inbound_accept_clear will decide whether to
 *    accept this in clear or not. In both the cases, if the returned
 *    message (IP header + 8 bytes) that caused the icmp message has
 *    AH/ESP headers, it is sent up to AH/ESP for validation before
 *    sending up. If there are only 8 bytes of returned message, then
 *    upper client will not be notified.
 * 3) Check with global policy to see whether it matches the constaints.
 *    But this will be done only if icmp_accept_messages_in_clear is
 *    zero.
 * 4) If we need to change both in IP and ULP, then the decision taken
 *    while affecting the values in IP and while delivering up to TCP
 *    should be the same.
 *
 *      There are two cases.
 *
 *      a) If we reject data at the IP layer (ipsec_check_global_policy()
 *         failed), we will not deliver it to the ULP, even though they
 *         are *willing* to accept in *clear*. This is fine as our global
 *         disposition to icmp messages asks us reject the datagram.
 *
 *      b) If we accept data at the IP layer (ipsec_check_global_policy()
 *         succeeded or icmp_accept_messages_in_clear is 1), and not able
 *         to deliver it to ULP (policy failed), it can lead to
 *         consistency problems. The cases known at this time are
 *         ICMP_DESTINATION_UNREACHABLE  messages with following code
 *         values :
 *
 *         - ICMP_FRAGMENTATION_NEEDED : IP adapts to the new value
 *           and Upper layer rejects. Then the communication will
 *           come to a stop. This is solved by making similar decisions
 *           at both levels. Currently, when we are unable to deliver
 *           to the Upper Layer (due to policy failures) while IP has
 *           adjusted dce_pmtu, the next outbound datagram would
 *           generate a local ICMP_FRAGMENTATION_NEEDED message - which
 *           will be with the right level of protection. Thus the right
 *           value will be communicated even if we are not able to
 *           communicate when we get from the wire initially. But this
```

```
 *           assumes there would be at least one outbound datagram after
 *           IP has adjusted its dce_pmtu value. To make things
 *           simpler, we accept in clear after the validation of
 *           AH/ESP headers.
 *
 *         - Other ICMP ERRORS : We may not be able to deliver it to the
 *           upper layer depending on the level of protection the upper
 *           layer expects and the disposition in ipsec_inbound_accept_clear().
 *           ipsec_inbound_accept_clear() decides whether a given ICMP error
 *           should be accepted in clear when the Upper layer expects secure.
 *           Thus the communication may get aborted by some bad ICMP
 *           packets.
 */
mblk_t *
icmp_inbound_v4(mblk_t *mp, ip_recv_attr_t *ira)
{
        icmph_t         *icmph;
        ipha_t          *ipha;          /* Outer header */
        int             ip_hdr_length;  /* Outer header length */
        boolean_t       interested;
        ipif_t          *ipif;
        uint32_t        ts;
        uint32_t        *tsp;
        timestruc_t     now;
        ill_t           *ill = ira->ira_ill;
        ip_stack_t      *ipst = ill->ill_ipst;
        zoneid_t        zoneid = ira->ira_zoneid;
        int             len_needed;
        mblk_t          *mp_ret = NULL;

        ipha = (ipha_t *)mp->b_rptr;

        BUMP_MIB(&ipst->ips_icmp_mib, icmpInMsgs);

        ip_hdr_length = ira->ira_ip_hdr_length;
        if ((mp->b_wptr - mp->b_rptr) < (ip_hdr_length + ICMPH_SIZE)) {
                if (ira->ira_pktlen < (ip_hdr_length + ICMPH_SIZE)) {
                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
                        ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
                        freemsg(mp);
                        return (NULL);
                }
                /* Last chance to get real. */
                ipha = ip_pullup(mp, ip_hdr_length + ICMPH_SIZE, ira);
                if (ipha == NULL) {
                        BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
                        freemsg(mp);
                        return (NULL);
                }
        }

        /* The IP header will always be a multiple of four bytes */
        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
        ip2dbg(("icmp_inbound_v4: type %d code %d\n", icmph->icmph_type,
            icmph->icmph_code));

        /*
         * We will set "interested" to "true" if we should pass a copy to
         * the transport or if we handle the packet locally.
         */
        interested = B_FALSE;
        switch (icmph->icmph_type) {
        case ICMP_ECHO_REPLY:
                BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchoReps);
                break;
        case ICMP_DEST_UNREACHABLE:
```



```
                if (icmph->icmph_code == ICMP_FRAGMENTATION_NEEDED)
                        BUMP_MIB(&ipst->ips_icmp_mib, icmpInFragNeeded);
                interested = B_TRUE;    /* Pass up to transport */
                BUMP_MIB(&ipst->ips_icmp_mib, icmpInDestUnreachs);
                break;
        case ICMP_SOURCE_QUENCH:
                interested = B_TRUE;    /* Pass up to transport */
                BUMP_MIB(&ipst->ips_icmp_mib, icmpInSrcQuenchs);
                break;
        case ICMP_REDIRECT:
                if (!ipst->ips_ip_ignore_redirect)
                        interested = B_TRUE;
                BUMP_MIB(&ipst->ips_icmp_mib, icmpInRedirects);
                break;
        case ICMP_ECHO_REQUEST:
                /*
                 * Whether to respond to echo requests that come in as IP
                 * broadcasts or as IP multicast is subject to debate
                 * (what isn't?).  We aim to please, you pick it.
                 * Default is do it.
                 */
                if (ira->ira_flags & IRAF_MULTICAST) {
                        /* multicast: respond based on tunable */
                        interested = ipst->ips_ip_g_resp_to_echo_mcast;
                } else if (ira->ira_flags & IRAF_BROADCAST) {
                        /* broadcast: respond based on tunable */
                        interested = ipst->ips_ip_g_resp_to_echo_bcast;
                } else {
                        /* unicast: always respond */
                        interested = B_TRUE;
                }
                BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchos);
                if (!interested) {
                        /* We never pass these to RAW sockets */
                        freemsg(mp);
                        return (NULL);
                }

                /* Check db_ref to make sure we can modify the packet. */
                if (mp->b_datap->db_ref > 1) {
                        mblk_t  *mp1;

                        mp1 = copymsg(mp);
                        freemsg(mp);
                        if (!mp1) {
                                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
                                return (NULL);
                        }
                        mp = mp1;
                        ipha = (ipha_t *)mp->b_rptr;
                        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
                }
                icmph->icmph_type = ICMP_ECHO_REPLY;
                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutEchoReps);
                icmp_send_reply_v4(mp, ipha, icmph, ira);
                return (NULL);

        case ICMP_ROUTER_ADVERTISEMENT:
        case ICMP_ROUTER_SOLICITATION:
                break;
        case ICMP_TIME_EXCEEDED:
                interested = B_TRUE;    /* Pass up to transport */
                BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimeExcds);
                break;
        case ICMP_PARAM_PROBLEM:
                interested = B_TRUE;    /* Pass up to transport */
```

```
1448                BUMP_MIB(&ipst->ips_icmp_mib, icmpInParmProbs);
1449                break;
1450        case ICMP_TIME_STAMP_REQUEST:
1451                /* Response to Time Stamp Requests is local policy. */
1452                if (ipst->ips_ip_g_resp_to_timestamp) {
1453                        if (ira->ira_flags & IRAF_MULTIBROADCAST)
1454                                interested =
1455                                    ipst->ips_ip_g_resp_to_timestamp_bcast;
1456                        else
1457                                interested = B_TRUE;
1458                }
1459                if (!interested) {
1460                        /* We never pass these to RAW sockets */
1461                        freemsg(mp);
1462                        return (NULL);
1463                }

1465                /* Make sure we have enough of the packet */
1466                len_needed = ip_hdr_length + ICMPH_SIZE +
1467                    3 * sizeof (uint32_t);

1469                if (mp->b_wptr - mp->b_rptr < len_needed) {
1470                        ipha = ip_pullup(mp, len_needed, ira);
1471                        if (ipha == NULL) {
1472                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1473                                ip_drop_input("ipIfStatsInDiscards - ip_pullup",
1474                                    mp, ill);
1475                                freemsg(mp);
1476                                return (NULL);
1477                        }
1478                        /* Refresh following the pullup. */
1479                        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1480                }
1481                BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimestamps);
1482                /* Check db_ref to make sure we can modify the packet. */
1483                if (mp->b_datap->db_ref > 1) {
1484                        mblk_t  *mp1;

1486                        mp1 = copymsg(mp);
1487                        freemsg(mp);
1488                        if (!mp1) {
1489                                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1490                                return (NULL);
1491                        }
1492                        mp = mp1;
1493                        ipha = (ipha_t *)mp->b_rptr;
1494                        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1495                }
1496                icmph->icmph_type = ICMP_TIME_STAMP_REPLY;
1497                tsp = (uint32_t *)&icmph[1];
1498                tsp++;          /* Skip past 'originate time' */
1499                /* Compute # of milliseconds since midnight */
1500                gethrestime(&now);
1501                ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
1502                    now.tv_nsec / (NANOSEC / MILLISEC);
1503                *tsp++ = htonl(ts);     /* Lay in 'receive time' */
1504                *tsp++ = htonl(ts);     /* Lay in 'send time' */
1505                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutTimestampReps);
1506                icmp_send_reply_v4(mp, ipha, icmph, ira);
1507                return (NULL);

1509        case ICMP_TIME_STAMP_REPLY:
1510                BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimestampReps);
1511                break;
1512        case ICMP_INFO_REQUEST:
1513                /* Per RFC 1122 3.2.2.7, ignore this. */
```



```
1514        case ICMP_INFO_REPLY:
1515                break;
1516        case ICMP_ADDRESS_MASK_REQUEST:
1517                if (ira->ira_flags & IRAF_MULTIBROADCAST) {
1518                        interested =
1519                            ipst->ips_ip_respond_to_address_mask_broadcast;
1520                } else {
1521                        interested = B_TRUE;
1522                }
1523                if (!interested) {
1524                        /* We never pass these to RAW sockets */
1525                        freemsg(mp);
1526                        return (NULL);
1527                }
1528                len_needed = ip_hdr_length + ICMPH_SIZE + IP_ADDR_LEN;
1529                if (mp->b_wptr - mp->b_rptr < len_needed) {
1530                        ipha = ip_pullup(mp, len_needed, ira);
1531                        if (ipha == NULL) {
1532                                BUMP_MIB(ill->ill_ip_mib,
1533                                    ipIfStatsInTruncatedPkts);
1534                                ip_drop_input("ipIfStatsInTruncatedPkts", mp,
1535                                    ill);
1536                                freemsg(mp);
1537                                return (NULL);
1538                        }
1539                        /* Refresh following the pullup. */
1540                        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1541                }
1542                BUMP_MIB(&ipst->ips_icmp_mib, icmpInAddrMasks);
1543                /* Check db_ref to make sure we can modify the packet. */
1544                if (mp->b_datap->db_ref > 1) {
1545                        mblk_t  *mp1;

1547                        mp1 = copymsg(mp);
1548                        freemsg(mp);
1549                        if (!mp1) {
1550                                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1551                                return (NULL);
1552                        }
1553                        mp = mp1;
1554                        ipha = (ipha_t *)mp->b_rptr;
1555                        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1556                }
1557                /*
1558                 * Need the ipif with the mask be the same as the source
1559                 * address of the mask reply. For unicast we have a specific
1560                 * ipif. For multicast/broadcast we only handle onlink
1561                 * senders, and use the source address to pick an ipif.
1562                 */
1563                ipif = ipif_lookup_addr(ipha->ipha_dst, ill, zoneid, ipst);
1564                if (ipif == NULL) {
1565                        /* Broadcast or multicast */
1566                        ipif = ipif_lookup_remote(ill, ipha->ipha_src, zoneid);
1567                        if (ipif == NULL) {
1568                                freemsg(mp);
1569                                return (NULL);
1570                        }
1571                }
1572                icmph->icmph_type = ICMP_ADDRESS_MASK_REPLY;
1573                bcopy(&ipif->ipif_net_mask, &icmph[1], IP_ADDR_LEN);
1574                ipif_refrele(ipif);
1575                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutAddrMaskReps);
1576                icmp_send_reply_v4(mp, ipha, icmph, ira);
1577                return (NULL);

1579        case ICMP_ADDRESS_MASK_REPLY:
```

```
1580                BUMP_MIB(&ipst->ips_icmp_mib, icmpInAddrMaskReps);
1581                break;
1582        default:
1583                interested = B_TRUE;    /* Pass up to transport */
1584                BUMP_MIB(&ipst->ips_icmp_mib, icmpInUnknowns);
1585                break;
1586        }
1587        /*
1588         * See if there is an ICMP client to avoid an extra copymsg/freemsg
1589         * if there isn't one.
1590         */
1591        if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_ICMP].connf_head != NULL) {
1592                /* If there is an ICMP client and we want one too, copy it. */

1594                if (!interested) {
1595                        /* Caller will deliver to RAW sockets */
1596                        return (mp);
1597                }
1598                mp_ret = copymsg(mp);
1599                if (mp_ret == NULL) {
1600                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1601                        ip_drop_input("ipIfStatsInDiscards - copymsg", mp, ill);
1602                }
1603        } else if (!interested) {
1604                /* Neither we nor raw sockets are interested. Drop packet now */
1605                freemsg(mp);
1606                return (NULL);
1607        }

1609        /*
1610         * ICMP error or redirect packet. Make sure we have enough of
1611         * the header and that db_ref == 1 since we might end up modifying
1612         * the packet.
1613         */
1614        if (mp->b_cont != NULL) {
1615                if (ip_pullup(mp, -1, ira) == NULL) {
1616                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1617                        ip_drop_input("ipIfStatsInDiscards - ip_pullup",
1618                            mp, ill);
1619                        freemsg(mp);
1620                        return (mp_ret);
1621                }
1622        }

1624        if (mp->b_datap->db_ref > 1) {
1625                mblk_t  *mp1;

1627                mp1 = copymsg(mp);
1628                if (mp1 == NULL) {
1629                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1630                        ip_drop_input("ipIfStatsInDiscards - copymsg", mp, ill);
1631                        freemsg(mp);
1632                        return (mp_ret);
1633                }
1634                freemsg(mp);
1635                mp = mp1;
1636        }

1638        /*
1639         * In case mp has changed, verify the message before any further
1640         * processes.
1641         */
1642        ipha = (ipha_t *)mp->b_rptr;
1643        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1644        if (!icmp_inbound_verify_v4(mp, icmph, ira)) {
1645                freemsg(mp);
```



```
1646                return (mp_ret);
1647        }

1649        switch (icmph->icmph_type) {
1650        case ICMP_REDIRECT:
1651                icmp_redirect_v4(mp, ipha, icmph, ira);
1652                break;
1653        case ICMP_DEST_UNREACHABLE:
1654                if (icmph->icmph_code == ICMP_FRAGMENTATION_NEEDED) {
1655                        /* Update DCE and adjust MTU is icmp header if needed */
1656                        icmp_inbound_too_big_v4(icmph, ira);
1657                }
1658                /* FALLTHRU */
1659        default:
1660                icmp_inbound_error_fanout_v4(mp, icmph, ira);
1661                break;
1662        }
1663        return (mp_ret);
1664 }

1666 /*
1667  * Send an ICMP echo, timestamp or address mask reply.
1668  * The caller has already updated the payload part of the packet.
1669  * We handle the ICMP checksum, IP source address selection and feed
1670  * the packet into ip_output_simple.
1671  */
1672 static void
1673 icmp_send_reply_v4(mblk_t *mp, ipha_t *ipha, icmph_t *icmph,
1674     ip_recv_attr_t *ira)
1675 {
1676        uint_t          ip_hdr_length = ira->ira_ip_hdr_length;
1677        ill_t           *ill = ira->ira_ill;
1678        ip_stack_t      *ipst = ill->ill_ipst;
1679        ip_xmit_attr_t  ixas;

1681        /* Send out an ICMP packet */
1682        icmph->icmph_checksum = 0;
1683        icmph->icmph_checksum = IP_CSUM(mp, ip_hdr_length, 0);
1684        /* Reset time to live. */
1685        ipha->ipha_ttl = ipst->ips_ip_def_ttl;
1686        {
1687                /* Swap source and destination addresses */
1688                ipaddr_t tmp;

1690                tmp = ipha->ipha_src;
1691                ipha->ipha_src = ipha->ipha_dst;
1692                ipha->ipha_dst = tmp;
1693        }
1694        ipha->ipha_ident = 0;
1695        if (!IS_SIMPLE_IPH(ipha))
1696                icmp_options_update(ipha);

1698        bzero(&ixas, sizeof (ixas));
1699        ixas.ixa_flags = IXAF_BASIC_SIMPLE_V4;
1700        ixas.ixa_zoneid = ira->ira_zoneid;
1701        ixas.ixa_cred = kcred;
1702        ixas.ixa_cpid = NOPID;
1703        ixas.ixa_tsl = ira->ira_tsl;    /* Behave as a multi-level responder */
1704        ixas.ixa_ifindex = 0;
1705        ixas.ixa_ipst = ipst;
1706        ixas.ixa_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;

1708        if (!(ira->ira_flags & IRAF_IPSEC_SECURE)) {
1709                /*
1710                 * This packet should go out the same way as it
1711                 * came in i.e in clear, independent of the IPsec policy
```

```
1712                 * for transmitting packets.
1713                 */
1714                ixas.ixa_flags |= IXAF_NO_IPSEC;
1715        } else {
1716                if (!ipsec_in_to_out(ira, &ixas, mp, ipha, NULL)) {
1717                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1718                        /* Note: mp already consumed and ip_drop_packet done */
1719                        return;
1720                }
1721        }
1722        if (ira->ira_flags & IRAF_MULTIBROADCAST) {
1723                /*
1724                 * Not one or our addresses (IRE_LOCALs), thus we let
1725                 * ip_output_simple pick the source.
1726                 */
1727                ipha->ipha_src = INADDR_ANY;
1728                ixas.ixa_flags |= IXAF_SET_SOURCE;
1729        }
1730        /* Should we send with DF and use dce_pmtu? */
1731        if (ipst->ips_ipv4_icmp_return_pmtu) {
1732                ixas.ixa_flags |= IXAF_PMTU_DISCOVERY;
1733                ipha->ipha_fragment_offset_and_flags |= IPH_DF_HTONS;
1734        }

1736        BUMP_MIB(&ipst->ips_icmp_mib, icmpOutMsgs);

1738        (void) ip_output_simple(mp, &ixas);
1739        ixa_cleanup(&ixas);
1740 }

1742 /*
1743  * Verify the ICMP messages for either for ICMP error or redirect packet.
1744  * The caller should have fully pulled up the message. If it's a redirect
1745  * packet, only basic checks on IP header will be done; otherwise, verify
1746  * the packet by looking at the included ULP header.
1747  *
1748  * Called before icmp_inbound_error_fanout_v4 is called.
1749  */
1750 static boolean_t
1751 icmp_inbound_verify_v4(mblk_t *mp, icmph_t *icmph, ip_recv_attr_t *ira)
1752 {
1753        ill_t           *ill = ira->ira_ill;
1754        int             hdr_length;
1755        ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
1756        conn_t          *connp;
1757        ipha_t          *ipha;  /* Inner IP header */

1759        ipha = (ipha_t *)&icmph[1];
1760        if ((uchar_t *)ipha + IP_SIMPLE_HDR_LENGTH > mp->b_wptr)
1761                goto truncated;

1763        hdr_length = IPH_HDR_LENGTH(ipha);

1765        if ((IPH_HDR_VERSION(ipha) != IPV4_VERSION))
1766                goto discard_pkt;

1768        if (hdr_length < sizeof (ipha_t))
1769                goto truncated;

1771        if ((uchar_t *)ipha + hdr_length > mp->b_wptr)
1772                goto truncated;

1774        /*
1775         * Stop here for ICMP_REDIRECT.
1776         */
1777        if (icmph->icmph_type == ICMP_REDIRECT)
```



```
1778                return (B_TRUE);

1780        /*
1781         * ICMP errors only.
1782         */
1783        switch (ipha->ipha_protocol) {
1784        case IPPROTO_UDP:
1785                /*
1786                 * Verify we have at least ICMP_MIN_TP_HDR_LEN bytes of
1787                 * transport header.
1788                 */
1789                if ((uchar_t *)ipha + hdr_length + ICMP_MIN_TP_HDR_LEN >
1790                    mp->b_wptr)
1791                        goto truncated;
1792                break;
1793        case IPPROTO_TCP: {
1794                tcpha_t         *tcpha;

1796                /*
1797                 * Verify we have at least ICMP_MIN_TP_HDR_LEN bytes of
1798                 * transport header.
1799                 */
1800                if ((uchar_t *)ipha + hdr_length + ICMP_MIN_TP_HDR_LEN >
1801                    mp->b_wptr)
1802                        goto truncated;

1804                tcpha = (tcpha_t *)((uchar_t *)ipha + hdr_length);
1805                connp = ipcl_tcp_lookup_reversed_ipv4(ipha, tcpha, TCPS_LISTEN,
1806                    ipst);
1807                if (connp == NULL)
1808                        goto discard_pkt;

1810                if ((connp->conn_verifyicmp != NULL) &&
1811                    !connp->conn_verifyicmp(connp, tcpha, icmph, NULL, ira)) {
1812                        CONN_DEC_REF(connp);
1813                        goto discard_pkt;
1814                }
1815                CONN_DEC_REF(connp);
1816                break;
1817        }
1818        case IPPROTO_SCTP:
1819                /*
1820                 * Verify we have at least ICMP_MIN_TP_HDR_LEN bytes of
1821                 * transport header.
1822                 */
1823                if ((uchar_t *)ipha + hdr_length + ICMP_MIN_TP_HDR_LEN >
1824                    mp->b_wptr)
1825                        goto truncated;
1826                break;
1827        case IPPROTO_ESP:
1828        case IPPROTO_AH:
1829                break;
1830        case IPPROTO_ENCAP:
1831                if ((uchar_t *)ipha + hdr_length + sizeof (ipha_t) >
1832                    mp->b_wptr)
1833                        goto truncated;
1834                break;
1835        default:
1836                break;
1837        }

1839        return (B_TRUE);

1841 discard_pkt:
1842        /* Bogus ICMP error. */
1843        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
```

```
1844         return (B_FALSE);

1846 truncated:
1847         /* We pulled up everthing already. Must be truncated */
1848         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
1849         ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
1850         return (B_FALSE);
1851 }

1853 /* Table from RFC 1191 */
1854 static int icmp_frag_size_table[] =
1855 { 32000, 17914, 8166, 4352, 2002, 1496, 1006, 508, 296, 68 };

1857 /*
1858  * Process received ICMP Packet too big.
1859  * Just handles the DCE create/update, including using the above table of
1860  * PMTU guesses. The caller is responsible for validating the packet before
1861  * passing it in and also to fanout the ICMP error to any matching transport
1862  * conns. Assumes the message has been fully pulled up and verified.
1863  *
1864  * Before getting here, the caller has called icmp_inbound_verify_v4()
1865  * that should have verified with ULP to prevent undoing the changes we're
1866  * going to make to DCE. For example, TCP might have verified that the packet
1867  * which generated error is in the send window.
1868  *
1869  * In some cases modified this MTU in the ICMP header packet; the caller
1870  * should pass to the matching ULP after this returns.
1871  */
1872 static void
1873 icmp_inbound_too_big_v4(icmph_t *icmph, ip_recv_attr_t *ira)
1874 {
1875         dce_t           *dce;
1876         int             old_mtu;
1877         int             mtu, orig_mtu;
1878         ipaddr_t        dst;
1879         boolean_t       disable_pmtud;
1880         ill_t           *ill = ira->ira_ill;
1881         ip_stack_t      *ipst = ill->ill_ipst;
1882         uint_t          hdr_length;
1883         ipha_t          *ipha;

1885         /* Caller already pulled up everything. */
1886         ipha = (ipha_t *)&icmph[1];
1887         ASSERT(icmph->icmph_type == ICMP_DEST_UNREACHABLE &&
1888             icmph->icmph_code == ICMP_FRAGMENTATION_NEEDED);
1889         ASSERT(ill != NULL);

1891         hdr_length = IPH_HDR_LENGTH(ipha);

1893         /*
1894          * We handle path MTU for source routed packets since the DCE
1895          * is looked up using the final destination.
1896          */
1897         dst = ip_get_dst(ipha);

1899         dce = dce_lookup_and_add_v4(dst, ipst);
1900         if (dce == NULL) {
1901                 /* Couldn't add a unique one - ENOMEM */
1902                 ip1dbg(("icmp_inbound_too_big_v4: no dce for 0x%x\n",
1903                     ntohl(dst)));
1904                 return;
1905         }

1907         /* Check for MTU discovery advice as described in RFC 1191 */
1908         mtu = ntohs(icmph->icmph_du_mtu);
1909         orig_mtu = mtu;
```



```
1910         disable_pmtud = B_FALSE;

1912         mutex_enter(&dce->dce_lock);
1913         if (dce->dce_flags & DCEF_PMTU)
1914                 old_mtu = dce->dce_pmtu;
1915         else
1916                 old_mtu = ill->ill_mtu;

1918         if (icmph->icmph_du_zero != 0 || mtu < ipst->ips_ip_pmtu_min) {
1919                 uint32_t length;
1920                 int     i;

1922                 /*
1923                  * Use the table from RFC 1191 to figure out
1924                  * the next "plateau" based on the length in
1925                  * the original IP packet.
1926                  */
1927                 length = ntohs(ipha->ipha_length);
1928                 DTRACE_PROBE2(ip4__pmtu__guess, dce_t *, dce,
1929                     uint32_t, length);
1930                 if (old_mtu <= length &&
1931                     old_mtu >= length - hdr_length) {
1932                         /*
1933                          * Handle broken BSD 4.2 systems that
1934                          * return the wrong ipha_length in ICMP
1935                          * errors.
1936                          */
1937                         ip1dbg(("Wrong mtu: sent %d, dce %d\n",
1938                             length, old_mtu));
1939                         length -= hdr_length;
1940                 }
1941                 for (i = 0; i < A_CNT(icmp_frag_size_table); i++) {
1942                         if (length > icmp_frag_size_table[i])
1943                                 break;
1944                 }
1945                 if (i == A_CNT(icmp_frag_size_table)) {
1946                         /* Smaller than IP_MIN_MTU! */
1947                         ip1dbg(("Too big for packet size %d\n",
1948                             length));
1949                         disable_pmtud = B_TRUE;
1950                         mtu = ipst->ips_ip_pmtu_min;
1951                 } else {
1952                         mtu = icmp_frag_size_table[i];
1953                         ip1dbg(("Calculated mtu %d, packet size %d, "
1954                             "before %d\n", mtu, length, old_mtu));
1955                         if (mtu < ipst->ips_ip_pmtu_min) {
1956                                 mtu = ipst->ips_ip_pmtu_min;
1957                                 disable_pmtud = B_TRUE;
1958                         }
1959                 }
1960         }
1961         if (disable_pmtud)
1962                 dce->dce_flags |= DCEF_TOO_SMALL_PMTU;
1963         else
1964                 dce->dce_flags &= ~DCEF_TOO_SMALL_PMTU;

1966         dce->dce_pmtu = MIN(old_mtu, mtu);
1967         /* Prepare to send the new max frag size for the ULP. */
1968         icmph->icmph_du_zero = 0;
1969         icmph->icmph_du_mtu =  htons((uint16_t)dce->dce_pmtu);
1970         DTRACE_PROBE4(ip4__pmtu__change, icmph_t *, icmph, dce_t *,
1971             dce, int, orig_mtu, int, mtu);

1973         /* We now have a PMTU for sure */
1974         dce->dce_flags |= DCEF_PMTU;
1975         dce->dce_last_change_time = TICK_TO_SEC(ddi_get_lbolt64());
```

```
1976         mutex_exit(&dce->dce_lock);
1977         /*
1978          * After dropping the lock the new value is visible to everyone.
1979          * Then we bump the generation number so any cached values reinspect
1980          * the dce_t.
1981          */
1982         dce_increment_generation(dce);
1983         dce_refrele(dce);
1984 }

1986 /*
1987  * If the packet in error is Self-Encapsulated, icmp_inbound_error_fanout_v4
1988  * calls this function.
1989  */
1990 static mblk_t *
1991 icmp_inbound_self_encap_error_v4(mblk_t *mp, ipha_t *ipha, ipha_t *in_ipha)
1992 {
1993         int length;

1995         ASSERT(mp->b_datap->db_type == M_DATA);

1997         /* icmp_inbound_v4 has already pulled up the whole error packet */
1998         ASSERT(mp->b_cont == NULL);

2000         /*
2001          * The length that we want to overlay is the inner header
2002          * and what follows it.
2003          */
2004         length = msgdsize(mp) - ((uchar_t *)in_ipha - mp->b_rptr);

2006         /*
2007          * Overlay the inner header and whatever follows it over the
2008          * outer header.
2009          */
2010         bcopy((uchar_t *)in_ipha, (uchar_t *)ipha, length);

2012         /* Adjust for what we removed */
2013         mp->b_wptr -= (uchar_t *)in_ipha - (uchar_t *)ipha;
2014         return (mp);
2015 }

2017 /*
2018  * Try to pass the ICMP message upstream in case the ULP cares.
2019  *
2020  * If the packet that caused the ICMP error is secure, we send
2021  * it to AH/ESP to make sure that the attached packet has a
2022  * valid association. ipha in the code below points to the
2023  * IP header of the packet that caused the error.
2024  *
2025  * For IPsec cases, we let the next-layer-up (which has access to
2026  * cached policy on the conn_t, or can query the SPD directly)
2027  * subtract out any IPsec overhead if they must.  We therefore make no
2028  * adjustments here for IPsec overhead.
2029  *
2030  * IFN could have been generated locally or by some router.
2031  *
2032  * LOCAL : ire_send_wire (before calling ipsec_out_process) can call
2033  * icmp_frag_needed/icmp_pkt2big_v6 to generated a local IFN.
2034  *         This happens because IP adjusted its value of MTU on an
2035  *         earlier IFN message and could not tell the upper layer,
2036  *         the new adjusted value of MTU e.g. Packet was encrypted
2037  *         or there was not enough information to fanout to upper
2038  *         layers. Thus on the next outbound datagram, ire_send_wire
2039  *         generates the IFN, where IPsec processing has *not* been
2040  *         done.
2041  *
```



```
2042  *          Note that we retain ixa_fragsize across IPsec thus once
2043  *          we have picking ixa_fragsize and entered ipsec_out_process we do
2044  *          no change the fragsize even if the path MTU changes before
2045  *          we reach ip_output_post_ipsec.
2046  *
2047  *          In the local case, IRAF_LOOPBACK will be set indicating
2048  *          that IFN was generated locally.
2049  *
2050  * ROUTER : IFN could be secure or non-secure.
2051  *
2052  *          * SECURE : We use the IPSEC_IN to fanout to AH/ESP if the
2053  *            packet in error has AH/ESP headers to validate the AH/ESP
2054  *            headers. AH/ESP will verify whether there is a valid SA or
2055  *            not and send it back. We will fanout again if we have more
2056  *            data in the packet.
2057  *
2058  *            If the packet in error does not have AH/ESP, we handle it
2059  *            like any other case.
2060  *
2061  *          * NON_SECURE : If the packet in error has AH/ESP headers, we send it
2062  *            up to AH/ESP for validation. AH/ESP will verify whether there is a
2063  *            valid SA or not and send it back. We will fanout again if
2064  *            we have more data in the packet.
2065  *
2066  *            If the packet in error does not have AH/ESP, we handle it
2067  *            like any other case.
2068  *
2069  * The caller must have called icmp_inbound_verify_v4.
2070  */
2071 static void
2072 icmp_inbound_error_fanout_v4(mblk_t *mp, icmph_t *icmph, ip_recv_attr_t *ira)
2073 {
2074         uint16_t        *up;    /* Pointer to ports in ULP header */
2075         uint32_t        ports;  /* reversed ports for fanout */
2076         ipha_t          ripha;  /* With reversed addresses */
2077         ipha_t          *ipha;  /* Inner IP header */
2078         uint_t          hdr_length; /* Inner IP header length */
2079         tcpha_t         *tcpha;
2080         conn_t          *connp;
2081         ill_t           *ill = ira->ira_ill;
2082         ip_stack_t      *ipst = ill->ill_ipst;
2083         ipsec_stack_t   *ipss = ipst->ips_netstack->netstack_ipsec;
2084         ill_t           *rill = ira->ira_rill;

2086         /* Caller already pulled up everything. */
2087         ipha = (ipha_t *)&icmph[1];
2088         ASSERT((uchar_t *)&ipha[1] <= mp->b_wptr);
2089         ASSERT(mp->b_cont == NULL);

2091         hdr_length = IPH_HDR_LENGTH(ipha);
2092         ira->ira_protocol = ipha->ipha_protocol;

2094         /*
2095          * We need a separate IP header with the source and destination
2096          * addresses reversed to do fanout/classification because the ipha in
2097          * the ICMP error is in the form we sent it out.
2098          */
2099         ripha.ipha_src = ipha->ipha_dst;
2100         ripha.ipha_dst = ipha->ipha_src;
2101         ripha.ipha_protocol = ipha->ipha_protocol;
2102         ripha.ipha_version_and_hdr_length = ipha->ipha_version_and_hdr_length;

2104         ip2dbg(("icmp_inbound_error_v4: proto %d %x to %x: %d/%d\n",
2105             ripha.ipha_protocol, ntohl(ipha->ipha_src),
2106             ntohl(ipha->ipha_dst),
2107             icmph->icmph_type, icmph->icmph_code));
```

```
2109         switch (ipha->ipha_protocol) {
2110         case IPPROTO_UDP:
2111                 up = (uint16_t *)((uchar_t *)ipha + hdr_length);
2112
2113                 /* Attempt to find a client stream based on port. */
2114                 ip2dbg(("icmp_inbound_error_v4: UDP ports %d to %d\n",
2115                     ntohs(up[0]), ntohs(up[1])));
2116
2117                 /* Note that we send error to all matches. */
2118                 ira->ira_flags |= IRAF_ICMP_ERROR;
2119                 ip_fanout_udp_multi_v4(mp, &ripha, up[0], up[1], ira);
2120                 ira->ira_flags &= ~IRAF_ICMP_ERROR;
2121                 return;
2122
2123         case IPPROTO_TCP:
2124                 /*
2125                  * Find a TCP client stream for this packet.
2126                  * Note that we do a reverse lookup since the header is
2127                  * in the form we sent it out.
2128                  */
2129                 tcpha = (tcpha_t *)((uchar_t *)ipha + hdr_length);
2130                 connp = ipcl_tcp_lookup_reversed_ipv4(ipha, tcpha, TCPS_LISTEN,
2131                     ipst);
2132                 if (connp == NULL)
2133                         goto discard_pkt;
2134
2135                 if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2136                     (ira->ira_flags & IRAF_IPSEC_SECURE)) {
2137                         mp = ipsec_check_inbound_policy(mp, connp,
2138                             ipha, NULL, ira);
2139                         if (mp == NULL) {
2140                                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2141                                 /* Note that mp is NULL */
2142                                 ip_drop_input("ipIfStatsInDiscards", mp, ill);
2143                                 CONN_DEC_REF(connp);
2144                                 return;
2145                         }
2146                 }
2147
2148                 ira->ira_flags |= IRAF_ICMP_ERROR;
2149                 ira->ira_ill = ira->ira_rill = NULL;
2150                 if (IPCL_IS_TCP(connp)) {
2151                         SQUEUE_ENTER_ONE(connp->conn_sqp, mp,
2152                             connp->conn_recvicmp, connp, ira, SQ_FILL,
2153                             SQTAG_TCP_INPUT_ICMP_ERR);
2154                 } else {
2155                         /* Not TCP; must be SOCK_RAW, IPPROTO_TCP */
2156                         (connp->conn_recv)(connp, mp, NULL, ira);
2157                         CONN_DEC_REF(connp);
2158                 }
2159                 ira->ira_ill = ill;
2160                 ira->ira_rill = rill;
2161                 ira->ira_flags &= ~IRAF_ICMP_ERROR;
2162                 return;
2163
2164         case IPPROTO_SCTP:
2165                 up = (uint16_t *)((uchar_t *)ipha + hdr_length);
2166                 /* Find a SCTP client stream for this packet. */
2167                 ((uint16_t *)&ports)[0] = up[1];
2168                 ((uint16_t *)&ports)[1] = up[0];
2169
2170                 ira->ira_flags |= IRAF_ICMP_ERROR;
2171                 ip_fanout_sctp(mp, &ripha, NULL, ports, ira);
2172                 ira->ira_flags &= ~IRAF_ICMP_ERROR;
2173                 return;
```



```
2175         case IPPROTO_ESP:
2176         case IPPROTO_AH:
2177                 if (!ipsec_loaded(ipss)) {
2178                         ip_proto_not_sup(mp, ira);
2179                         return;
2180                 }
2181
2182                 if (ipha->ipha_protocol == IPPROTO_ESP)
2183                         mp = ipsecesp_icmp_error(mp, ira);
2184                 else
2185                         mp = ipsecah_icmp_error(mp, ira);
2186                 if (mp == NULL)
2187                         return;
2188
2189                 /* Just in case ipsec didn't preserve the NULL b_cont */
2190                 if (mp->b_cont != NULL) {
2191                         if (!pullupmsg(mp, -1))
2192                                 goto discard_pkt;
2193                 }
2194
2195                 /*
2196                  * Note that ira_pktlen and ira_ip_hdr_length are no longer
2197                  * correct, but we don't use them any more here.
2198                  *
2199                  * If succesful, the mp has been modified to not include
2200                  * the ESP/AH header so we can fanout to the ULP's icmp
2201                  * error handler.
2202                  */
2203                 if (mp->b_wptr - mp->b_rptr < IP_SIMPLE_HDR_LENGTH)
2204                         goto truncated;
2205
2206                 /* Verify the modified message before any further processes. */
2207                 ipha = (ipha_t *)mp->b_rptr;
2208                 hdr_length = IPH_HDR_LENGTH(ipha);
2209                 icmph = (icmph_t *)&mp->b_rptr[hdr_length];
2210                 if (!icmp_inbound_verify_v4(mp, icmph, ira)) {
2211                         freemsg(mp);
2212                         return;
2213                 }
2214
2215                 icmp_inbound_error_fanout_v4(mp, icmph, ira);
2216                 return;
2217
2218         case IPPROTO_ENCAP: {
2219                 /* Look for self-encapsulated packets that caused an error */
2220                 ipha_t *in_ipha;
2221
2222                 /*
2223                  * Caller has verified that length has to be
2224                  * at least the size of IP header.
2225                  */
2226                 ASSERT(hdr_length >= sizeof (ipha_t));
2227                 /*
2228                  * Check the sanity of the inner IP header like
2229                  * we did for the outer header.
2230                  */
2231                 in_ipha = (ipha_t *)((uchar_t *)ipha + hdr_length);
2232                 if ((IPH_HDR_VERSION(in_ipha) != IPV4_VERSION)) {
2233                         goto discard_pkt;
2234                 }
2235                 if (IPH_HDR_LENGTH(in_ipha) < sizeof (ipha_t)) {
2236                         goto discard_pkt;
2237                 }
2238                 /* Check for Self-encapsulated tunnels */
2239                 if (in_ipha->ipha_src == ipha->ipha_src &&
```

```
2240                    in_ipha->ipha_dst == ipha->ipha_dst) {

2242                        mp = icmp_inbound_self_encap_error_v4(mp, ipha,
2243                            in_ipha);
2244                        if (mp == NULL)
2245                                goto discard_pkt;

2247                        /*
2248                         * Just in case self_encap didn't preserve the NULL
2249                         * b_cont
2250                         */
2251                        if (mp->b_cont != NULL) {
2252                                if (!pullupmsg(mp, -1))
2253                                        goto discard_pkt;
2254                        }
2255                        /*
2256                         * Note that ira_pktlen and ira_ip_hdr_length are no
2257                         * longer correct, but we don't use them any more here.
2258                         */
2259                        if (mp->b_wptr - mp->b_rptr < IP_SIMPLE_HDR_LENGTH)
2260                                goto truncated;

2262                        /*
2263                         * Verify the modified message before any further
2264                         * processes.
2265                         */
2266                        ipha = (ipha_t *)mp->b_rptr;
2267                        hdr_length = IPH_HDR_LENGTH(ipha);
2268                        icmph = (icmph_t *)&mp->b_rptr[hdr_length];
2269                        if (!icmp_inbound_verify_v4(mp, icmph, ira)) {
2270                                freemsg(mp);
2271                                return;
2272                        }

2274                        /*
2275                         * The packet in error is self-encapsualted.
2276                         * And we are finding it further encapsulated
2277                         * which we could not have possibly generated.
2278                         */
2279                        if (ipha->ipha_protocol == IPPROTO_ENCAP) {
2280                                goto discard_pkt;
2281                        }
2282                        icmp_inbound_error_fanout_v4(mp, icmph, ira);
2283                        return;
2284                }
2285                /* No self-encapsulated */
2286                /* FALLTHRU */
2287        }
2288        case IPPROTO_IPV6:
2289                if ((connp = ipcl_iptun_classify_v4(&ripha.ipha_src,
2290                    &ripha.ipha_dst, ipst)) != NULL) {
2291                        ira->ira_flags |= IRAF_ICMP_ERROR;
2292                        connp->conn_recvicmp(connp, mp, NULL, ira);
2293                        CONN_DEC_REF(connp);
2294                        ira->ira_flags &= ~IRAF_ICMP_ERROR;
2295                        return;
2296                }
2297                /*
2298                 * No IP tunnel is interested, fallthrough and see
2299                 * if a raw socket will want it.
2300                 */
2301                /* FALLTHRU */
2302        default:
2303                ira->ira_flags |= IRAF_ICMP_ERROR;
2304                ip_fanout_proto_v4(mp, &ripha, ira);
2305                ira->ira_flags &= ~IRAF_ICMP_ERROR;
```



```
2306                    return;
2307            }
2308            /* NOTREACHED */
2309    discard_pkt:
2310            BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2311            ip1dbg(("icmp_inbound_error_fanout_v4: drop pkt\n"));
2312            ip_drop_input("ipIfStatsInDiscards", mp, ill);
2313            freemsg(mp);
2314            return;

2316    truncated:
2317            /* We pulled up everthing already. Must be truncated */
2318            BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
2319            ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
2320            freemsg(mp);
2321    }

2323    /*
2324     * Common IP options parser.
2325     *
2326     * Setup routine: fill in *optp with options-parsing state, then
2327     * tail-call ipoptp_next to return the first option.
2328     */
2329    uint8_t
2330    ipoptp_first(ipoptp_t *optp, ipha_t *ipha)
2331    {
2332            uint32_t totallen; /* total length of all options */

2334            totallen = ipha->ipha_version_and_hdr_length -
2335                (uint8_t)((IP_VERSION << 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);
2336            totallen <<= 2;
2337            optp->ipoptp_next = (uint8_t *)(&ipha[1]);
2338            optp->ipoptp_end = optp->ipoptp_next + totallen;
2339            optp->ipoptp_flags = 0;
2340            return (ipoptp_next(optp));
2341    }

2343    /* Like above but without an ipha_t */
2344    uint8_t
2345    ipoptp_first2(ipoptp_t *optp, uint32_t totallen, uint8_t *opt)
2346    {
2347            optp->ipoptp_next = opt;
2348            optp->ipoptp_end = optp->ipoptp_next + totallen;
2349            optp->ipoptp_flags = 0;
2350            return (ipoptp_next(optp));
2351    }

2353    /*
2354     * Common IP options parser: extract next option.
2355     */
2356    uint8_t
2357    ipoptp_next(ipoptp_t *optp)
2358    {
2359            uint8_t *end = optp->ipoptp_end;
2360            uint8_t *cur = optp->ipoptp_next;
2361            uint8_t opt, len, pointer;

2363            /*
2364             * If cur > end already, then the ipoptp_end or ipoptp_next pointer
2365             * has been corrupted.
2366             */
2367            ASSERT(cur <= end);

2369            if (cur == end)
2370                    return (IPOPT_EOL);
```

```
2372         opt = cur[IPOPT_OPTVAL];
2373
2374         /*
2375          * Skip any NOP options.
2376          */
2377         while (opt == IPOPT_NOP) {
2378                 cur++;
2379                 if (cur == end)
2380                         return (IPOPT_EOL);
2381                 opt = cur[IPOPT_OPTVAL];
2382         }
2383
2384         if (opt == IPOPT_EOL)
2385                 return (IPOPT_EOL);
2386
2387         /*
2388          * Option requiring a length.
2389          */
2390         if ((cur + 1) >= end) {
2391                 optp->ipoptp_flags |= IPOPTP_ERROR;
2392                 return (IPOPT_EOL);
2393         }
2394         len = cur[IPOPT_OLEN];
2395         if (len < 2) {
2396                 optp->ipoptp_flags |= IPOPTP_ERROR;
2397                 return (IPOPT_EOL);
2398         }
2399         optp->ipoptp_cur = cur;
2400         optp->ipoptp_len = len;
2401         optp->ipoptp_next = cur + len;
2402         if (cur + len > end) {
2403                 optp->ipoptp_flags |= IPOPTP_ERROR;
2404                 return (IPOPT_EOL);
2405         }
2406
2407         /*
2408          * For the options which require a pointer field, make sure
2409          * its there, and make sure it points to either something
2410          * inside this option, or the end of the option.
2411          */
2412         switch (opt) {
2413         case IPOPT_RR:
2414         case IPOPT_TS:
2415         case IPOPT_LSRR:
2416         case IPOPT_SSRR:
2417                 if (len <= IPOPT_OFFSET) {
2418                         optp->ipoptp_flags |= IPOPTP_ERROR;
2419                         return (opt);
2420                 }
2421                 pointer = cur[IPOPT_OFFSET];
2422                 if (pointer - 1 > len) {
2423                         optp->ipoptp_flags |= IPOPTP_ERROR;
2424                         return (opt);
2425                 }
2426                 break;
2427         }
2428
2429         /*
2430          * Sanity check the pointer field based on the type of the
2431          * option.
2432          */
2433         switch (opt) {
2434         case IPOPT_RR:
2435         case IPOPT_SSRR:
2436         case IPOPT_LSRR:
2437                 if (pointer < IPOPT_MINOFF_SR)
```



```
2438                         optp->ipoptp_flags |= IPOPTP_ERROR;
2439                 break;
2440         case IPOPT_TS:
2441                 if (pointer < IPOPT_MINOFF_IT)
2442                         optp->ipoptp_flags |= IPOPTP_ERROR;
2443                 /*
2444                  * Note that the Internet Timestamp option also
2445                  * contains two four bit fields (the Overflow field,
2446                  * and the Flag field), which follow the pointer
2447                  * field.  We don't need to check that these fields
2448                  * fall within the length of the option because this
2449                  * was implicitely done above.  We've checked that the
2450                  * pointer value is at least IPOPT_MINOFF_IT, and that
2451                  * it falls within the option.  Since IPOPT_MINOFF_IT >
2452                  * IPOPT_POS_OV_FLG, we don't need the explicit check.
2453                  */
2454                 ASSERT(len > IPOPT_POS_OV_FLG);
2455                 break;
2456         }
2457
2458         return (opt);
2459 }
2460
2461 /*
2462  * Use the outgoing IP header to create an IP_OPTIONS option the way
2463  * it was passed down from the application.
2464  *
2465  * This is compatible with BSD in that it returns
2466  * the reverse source route with the final destination
2467  * as the last entry. The first 4 bytes of the option
2468  * will contain the final destination.
2469  */
2470 int
2471 ip_opt_get_user(conn_t *connp, uchar_t *buf)
2472 {
2473         ipoptp_t        opts;
2474         uchar_t         *opt;
2475         uint8_t         optval;
2476         uint8_t         optlen;
2477         uint32_t        len = 0;
2478         uchar_t         *buf1 = buf;
2479         uint32_t        totallen;
2480         ipaddr_t        dst;
2481         ip_pkt_t        *ipp = &connp->conn_xmit_ipp;
2482
2483         if (!(ipp->ipp_fields & IPPF_IPV4_OPTIONS))
2484                 return (0);
2485
2486         totallen = ipp->ipp_ipv4_options_len;
2487         if (totallen & 0x3)
2488                 return (0);
2489
2490         buf += IP_ADDR_LEN;     /* Leave room for final destination */
2491         len += IP_ADDR_LEN;
2492         bzero(buf1, IP_ADDR_LEN);
2493
2494         dst = connp->conn_faddr_v4;
2495
2496         for (optval = ipoptp_first2(&opts, totallen, ipp->ipp_ipv4_options);
2497             optval != IPOPT_EOL;
2498             optval = ipoptp_next(&opts)) {
2499                 int     off;
2500
2501                 opt = opts.ipoptp_cur;
2502                 if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
2503                         break;
```

```
2504                }
2505                optlen = opts.ipoptp_len;

2507                switch (optval) {
2508                case IPOPT_SSRR:
2509                case IPOPT_LSRR:

2511                        /*
2512                         * Insert destination as the first entry in the source
2513                         * route and move down the entries on step.
2514                         * The last entry gets placed at buf1.
2515                         */
2516                        buf[IPOPT_OPTVAL] = optval;
2517                        buf[IPOPT_OLEN] = optlen;
2518                        buf[IPOPT_OFFSET] = optlen;

2520                        off = optlen - IP_ADDR_LEN;
2521                        if (off < 0) {
2522                                /* No entries in source route */
2523                                break;
2524                        }
2525                        /* Last entry in source route if not already set */
2526                        if (dst == INADDR_ANY)
2527                                bcopy(opt + off, buf1, IP_ADDR_LEN);
2528                        off -= IP_ADDR_LEN;

2530                        while (off > 0) {
2531                                bcopy(opt + off,
2532                                    buf + off + IP_ADDR_LEN,
2533                                    IP_ADDR_LEN);
2534                                off -= IP_ADDR_LEN;
2535                        }
2536                        /* ipha_dst into first slot */
2537                        bcopy(&dst, buf + off + IP_ADDR_LEN,
2538                            IP_ADDR_LEN);
2539                        buf += optlen;
2540                        len += optlen;
2541                        break;

2543                default:
2544                        bcopy(opt, buf, optlen);
2545                        buf += optlen;
2546                        len += optlen;
2547                        break;
2548                }
2549        }
2550 done:
2551        /* Pad the resulting options */
2552        while (len & 0x3) {
2553                *buf++ = IPOPT_EOL;
2554                len++;
2555        }
2556        return (len);
2557 }

2559 /*
2560  * Update any record route or timestamp options to include this host.
2561  * Reverse any source route option.
2562  * This routine assumes that the options are well formed i.e. that they
2563  * have already been checked.
2564  */
2565 static void
2566 icmp_options_update(ipha_t *ipha)
2567 {
2568        ipoptp_t        opts;
2569        uchar_t         *opt;
```



```
2570        uint8_t         optval;
2571        ipaddr_t        src;            /* Our local address */
2572        ipaddr_t        dst;

2574        ip2dbg(("icmp_options_update\n"));
2575        src = ipha->ipha_src;
2576        dst = ipha->ipha_dst;

2578        for (optval = ipoptp_first(&opts, ipha);
2579            optval != IPOPT_EOL;
2580            optval = ipoptp_next(&opts)) {
2581                ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
2582                opt = opts.ipoptp_cur;
2583                ip2dbg(("icmp_options_update: opt %d, len %d\n",
2584                    optval, opts.ipoptp_len));
2585                switch (optval) {
2586                        int off1, off2;
2587                case IPOPT_SSRR:
2588                case IPOPT_LSRR:
2589                        /*
2590                         * Reverse the source route.  The first entry
2591                         * should be the next to last one in the current
2592                         * source route (the last entry is our address).
2593                         * The last entry should be the final destination.
2594                         */
2595                        off1 = IPOPT_MINOFF_SR - 1;
2596                        off2 = opt[IPOPT_OFFSET] - IP_ADDR_LEN - 1;
2597                        if (off2 < 0) {
2598                                /* No entries in source route */
2599                                ip1dbg((
2600                                    "icmp_options_update: bad src route\n"));
2601                                break;
2602                        }
2603                        bcopy((char *)opt + off2, &dst, IP_ADDR_LEN);
2604                        bcopy(&ipha->ipha_dst, (char *)opt + off2, IP_ADDR_LEN);
2605                        bcopy(&dst, &ipha->ipha_dst, IP_ADDR_LEN);
2606                        off2 -= IP_ADDR_LEN;

2608                        while (off1 < off2) {
2609                                bcopy((char *)opt + off1, &src, IP_ADDR_LEN);
2610                                bcopy((char *)opt + off2, (char *)opt + off1,
2611                                    IP_ADDR_LEN);
2612                                bcopy(&src, (char *)opt + off2, IP_ADDR_LEN);
2613                                off1 += IP_ADDR_LEN;
2614                                off2 -= IP_ADDR_LEN;
2615                        }
2616                        opt[IPOPT_OFFSET] = IPOPT_MINOFF_SR;
2617                        break;
2618                }
2619        }
2620 }

2622 /*
2623  * Process received ICMP Redirect messages.
2624  * Assumes the caller has verified that the headers are in the pulled up mblk.
2625  * Consumes mp.
2626  */
2627 static void
2628 icmp_redirect_v4(mblk_t *mp, ipha_t *ipha, icmph_t *icmph, ip_recv_attr_t *ira)
2629 {
2630        ire_t           *ire, *nire;
2631        ire_t           *prev_ire;
2632        ipaddr_t        src, dst, gateway;
2633        ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
2634        ipha_t          *inner_ipha;    /* Inner IP header */
```

```
2636         /* Caller already pulled up everything. */
2637         inner_ipha = (ipha_t *)&icmph[1];
2638         src = ipha->ipha_src;
2639         dst = inner_ipha->ipha_dst;
2640         gateway = icmph->icmph_rd_gateway;
2641         /* Make sure the new gateway is reachable somehow. */
2642         ire = ire_ftable_lookup_v4(gateway, 0, 0, IRE_ONLINK, NULL,
2643             ALL_ZONES, NULL, MATCH_IRE_TYPE, 0, ipst, NULL);
2644         /*
2645          * Make sure we had a route for the dest in question and that
2646          * that route was pointing to the old gateway (the source of the
2647          * redirect packet.)
2648          * We do longest match and then compare ire_gateway_addr below.
2649          */
2650         prev_ire = ire_ftable_lookup_v4(dst, 0, 0, 0, NULL, ALL_ZONES,
2651             NULL, MATCH_IRE_DSTONLY, 0, ipst, NULL);
2652         /*
2653          * Check that
2654          *      the redirect was not from ourselves
2655          *      the new gateway and the old gateway are directly reachable
2656          */
2657         if (prev_ire == NULL || ire == NULL ||
2658             (prev_ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK)) ||
2659             (prev_ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) ||
2660             !(ire->ire_type & IRE_IF_ALL) ||
2661             prev_ire->ire_gateway_addr != src) {
2662                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInBadRedirects);
2663                 ip_drop_input("icmpInBadRedirects - ire", mp, ira->ira_ill);
2664                 freemsg(mp);
2665                 if (ire != NULL)
2666                         ire_refrele(ire);
2667                 if (prev_ire != NULL)
2668                         ire_refrele(prev_ire);
2669                 return;
2670         }

2672         ire_refrele(prev_ire);
2673         ire_refrele(ire);

2675         /*
2676          * TODO: more precise handling for cases 0, 2, 3, the latter two
2677          * require TOS routing
2678          */
2679         switch (icmph->icmph_code) {
2680         case 0:
2681         case 1:
2682                 /* TODO: TOS specificity for cases 2 and 3 */
2683         case 2:
2684         case 3:
2685                 break;
2686         default:
2687                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInBadRedirects);
2688                 ip_drop_input("icmpInBadRedirects - code", mp, ira->ira_ill);
2689                 freemsg(mp);
2690                 return;
2691         }
2692         /*
2693          * Create a Route Association.  This will allow us to remember that
2694          * someone we believe told us to use the particular gateway.
2695          */
2696         ire = ire_create(
2697             (uchar_t *)&dst,                    /* dest addr */
2698             (uchar_t *)&ip_g_all_ones,          /* mask */
2699             (uchar_t *)&gateway,                /* gateway addr */
2700             IRE_HOST,
2701             NULL,                               /* ill */
```



```
2702             ALL_ZONES,
2703             (RTF_DYNAMIC | RTF_GATEWAY | RTF_HOST),
2704             NULL,                                   /* tsol_gc_t */
2705             ipst);

2707         if (ire == NULL) {
2708                 freemsg(mp);
2709                 return;
2710         }
2711         nire = ire_add(ire);
2712         /* Check if it was a duplicate entry */
2713         if (nire != NULL && nire != ire) {
2714                 ASSERT(nire->ire_identical_ref > 1);
2715                 ire_delete(nire);
2716                 ire_refrele(nire);
2717                 nire = NULL;
2718         }
2719         ire = nire;
2720         if (ire != NULL) {
2721                 ire_refrele(ire);               /* Held in ire_add */

2723                 /* tell routing sockets that we received a redirect */
2724                 ip_rts_change(RTM_REDIRECT, dst, gateway, IP_HOST_MASK, 0, src,
2725                     (RTF_DYNAMIC | RTF_GATEWAY | RTF_HOST), 0,
2726                     (RTA_DST | RTA_GATEWAY | RTA_NETMASK | RTA_AUTHOR), ipst);
2727         }

2729         /*
2730          * Delete any existing IRE_HOST type redirect ires for this destination.
2731          * This together with the added IRE has the effect of
2732          * modifying an existing redirect.
2733          */
2734         prev_ire = ire_ftable_lookup_v4(dst, 0, src, IRE_HOST, NULL,
2735             ALL_ZONES, NULL, (MATCH_IRE_GW | MATCH_IRE_TYPE), 0, ipst, NULL);
2736         if (prev_ire != NULL) {
2737                 if (prev_ire ->ire_flags & RTF_DYNAMIC)
2738                         ire_delete(prev_ire);
2739                 ire_refrele(prev_ire);
2740         }

2742         freemsg(mp);
2743 }

2745 /*
2746  * Generate an ICMP parameter problem message.
2747  * When called from ip_output side a minimal ip_recv_attr_t needs to be
2748  * constructed by the caller.
2749  */
2750 static void
2751 icmp_param_problem(mblk_t *mp, uint8_t ptr, ip_recv_attr_t *ira)
2752 {
2753         icmph_t icmph;
2754         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;

2756         mp = icmp_pkt_err_ok(mp, ira);
2757         if (mp == NULL)
2758                 return;

2760         bzero(&icmph, sizeof (icmph_t));
2761         icmph.icmph_type = ICMP_PARAM_PROBLEM;
2762         icmph.icmph_pp_ptr = ptr;
2763         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutParmProbs);
2764         icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
2765 }

2767 /*
```

```
 * Build and ship an IPv4 ICMP message using the packet data in mp, and
 * the ICMP header pointed to by "stuff".  (May be called as writer.)
 * Note: assumes that icmp_pkt_err_ok has been called to verify that
 * an icmp error packet can be sent.
 * Assigns an appropriate source address to the packet. If ipha_dst is
 * one of our addresses use it for source. Otherwise let ip_output_simple
 * pick the source address.
 */
static void
icmp_pkt(mblk_t *mp, void *stuff, size_t len, ip_recv_attr_t *ira)
{
        ipaddr_t dst;
        icmph_t *icmph;
        ipha_t  *ipha;
        uint_t  len_needed;
        size_t  msg_len;
        mblk_t  *mp1;
        ipaddr_t src;
        ire_t   *ire;
        ip_xmit_attr_t ixas;
        ip_stack_t *ipst = ira->ira_ill->ill_ipst;

        ipha = (ipha_t *)mp->b_rptr;

        bzero(&ixas, sizeof (ixas));
        ixas.ixa_flags = IXAF_BASIC_SIMPLE_V4;
        ixas.ixa_zoneid = ira->ira_zoneid;
        ixas.ixa_ifindex = 0;
        ixas.ixa_ipst = ipst;
        ixas.ixa_cred = kcred;
        ixas.ixa_cpid = NOPID;
        ixas.ixa_tsl = ira->ira_tsl;    /* Behave as a multi-level responder */
        ixas.ixa_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;

        if (ira->ira_flags & IRAF_IPSEC_SECURE) {
                /*
                 * Apply IPsec based on how IPsec was applied to
                 * the packet that had the error.
                 *
                 * If it was an outbound packet that caused the ICMP
                 * error, then the caller will have setup the IRA
                 * appropriately.
                 */
                if (!ipsec_in_to_out(ira, &ixas, mp, ipha, NULL)) {
                        BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsOutDiscards);
                        /* Note: mp already consumed and ip_drop_packet done */
                        return;
                }
        } else {
                /*
                 * This is in clear. The icmp message we are building
                 * here should go out in clear, independent of our policy.
                 */
                ixas.ixa_flags |= IXAF_NO_IPSEC;
        }

        /* Remember our eventual destination */
        dst = ipha->ipha_src;

        /*
         * If the packet was for one of our unicast addresses, make
         * sure we respond with that as the source. Otherwise
         * have ip_output_simple pick the source address.
         */
        ire = ire_ftable_lookup_v4(ipha->ipha_dst, 0, 0,
            (IRE_LOCAL|IRE_LOOPBACK), NULL, ira->ira_zoneid, NULL,
```



```
            MATCH_IRE_TYPE|MATCH_IRE_ZONEONLY, 0, ipst, NULL);
        if (ire != NULL) {
                ire_refrele(ire);
                src = ipha->ipha_dst;
        } else {
                src = INADDR_ANY;
                ixas.ixa_flags |= IXAF_SET_SOURCE;
        }

        /*
         * Check if we can send back more then 8 bytes in addition to
         * the IP header.  We try to send 64 bytes of data and the internal
         * header in the special cases of ipv4 encapsulated ipv4 or ipv6.
         */
        len_needed = IPH_HDR_LENGTH(ipha);
        if (ipha->ipha_protocol == IPPROTO_ENCAP ||
            ipha->ipha_protocol == IPPROTO_IPV6) {
                if (!pullupmsg(mp, -1)) {
                        BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsOutDiscards);
                        ip_drop_output("ipIfStatsOutDiscards", mp, NULL);
                        freemsg(mp);
                        return;
                }
                ipha = (ipha_t *)mp->b_rptr;

                if (ipha->ipha_protocol == IPPROTO_ENCAP) {
                        len_needed += IPH_HDR_LENGTH(((uchar_t *)ipha +
                            len_needed));
                } else {
                        ip6_t *ip6h = (ip6_t *)((uchar_t *)ipha + len_needed);

                        ASSERT(ipha->ipha_protocol == IPPROTO_IPV6);
                        len_needed += ip_hdr_length_v6(mp, ip6h);
                }
        }
        len_needed += ipst->ips_ip_icmp_return;
        msg_len = msgdsize(mp);
        if (msg_len > len_needed) {
                (void) adjmsg(mp, len_needed - msg_len);
                msg_len = len_needed;
        }
        mp1 = allocb(sizeof (icmp_ipha) + len, BPRI_MED);
        if (mp1 == NULL) {
                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutErrors);
                freemsg(mp);
                return;
        }
        mp1->b_cont = mp;
        mp = mp1;

        /*
         * Set IXAF_TRUSTED_ICMP so we can let the ICMP messages this
         * node generates be accepted in peace by all on-host destinations.
         * If we do NOT assume that all on-host destinations trust
         * self-generated ICMP messages, then rework here, ip6.c, and spd.c.
         * (Look for IXAF_TRUSTED_ICMP).
         */
        ixas.ixa_flags |= IXAF_TRUSTED_ICMP;

        ipha = (ipha_t *)mp->b_rptr;
        mp1->b_wptr = (uchar_t *)ipha + (sizeof (icmp_ipha) + len);
        *ipha = icmp_ipha;
        ipha->ipha_src = src;
        ipha->ipha_dst = dst;
        ipha->ipha_ttl = ipst->ips_ip_def_ttl;
        msg_len += sizeof (icmp_ipha) + len;
```

```
2900         if (msg_len > IP_MAXPACKET) {
2901                 (void) adjmsg(mp, IP_MAXPACKET - msg_len);
2902                 msg_len = IP_MAXPACKET;
2903         }
2904         ipha->ipha_length = htons((uint16_t)msg_len);
2905         icmph = (icmph_t *)&ipha[1];
2906         bcopy(stuff, icmph, len);
2907         icmph->icmph_checksum = 0;
2908         icmph->icmph_checksum = IP_CSUM(mp, (int32_t)sizeof (ipha_t), 0);
2909         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutMsgs);

2911         (void) ip_output_simple(mp, &ixas);
2912         ixa_cleanup(&ixas);
2913 }

2915 /*
2916  * Determine if an ICMP error packet can be sent given the rate limit.
2917  * The limit consists of an average frequency (icmp_pkt_err_interval measured
2918  * in milliseconds) and a burst size. Burst size number of packets can
2919  * be sent arbitrarely closely spaced.
2920  * The state is tracked using two variables to implement an approximate
2921  * token bucket filter:
2922  *      icmp_pkt_err_last - lbolt value when the last burst started
2923  *      icmp_pkt_err_sent - number of packets sent in current burst
2924  */
2925 boolean_t
2926 icmp_err_rate_limit(ip_stack_t *ipst)
2927 {
2928         clock_t now = TICK_TO_MSEC(ddi_get_lbolt());
2929         uint_t refilled; /* Number of packets refilled in tbf since last */
2930         /* Guard against changes by loading into local variable */
2931         uint_t err_interval = ipst->ips_ip_icmp_err_interval;

2933         if (err_interval == 0)
2934                 return (B_FALSE);

2936         if (ipst->ips_icmp_pkt_err_last > now) {
2937                 /* 100HZ lbolt in ms for 32bit arch wraps every 49.7 days */
2938                 ipst->ips_icmp_pkt_err_last = 0;
2939                 ipst->ips_icmp_pkt_err_sent = 0;
2940         }
2941         /*
2942          * If we are in a burst update the token bucket filter.
2943          * Update the "last" time to be close to "now" but make sure
2944          * we don't loose precision.
2945          */
2946         if (ipst->ips_icmp_pkt_err_sent != 0) {
2947                 refilled = (now - ipst->ips_icmp_pkt_err_last)/err_interval;
2948                 if (refilled > ipst->ips_icmp_pkt_err_sent) {
2949                         ipst->ips_icmp_pkt_err_sent = 0;
2950                 } else {
2951                         ipst->ips_icmp_pkt_err_sent -= refilled;
2952                         ipst->ips_icmp_pkt_err_last += refilled * err_interval;
2953                 }
2954         }
2955         if (ipst->ips_icmp_pkt_err_sent == 0) {
2956                 /* Start of new burst */
2957                 ipst->ips_icmp_pkt_err_last = now;
2958         }
2959         if (ipst->ips_icmp_pkt_err_sent < ipst->ips_ip_icmp_err_burst) {
2960                 ipst->ips_icmp_pkt_err_sent++;
2961                 ip1dbg(("icmp_err_rate_limit: %d sent in burst\n",
2962                     ipst->ips_icmp_pkt_err_sent));
2963                 return (B_FALSE);
2964         }
2965         ip1dbg(("icmp_err_rate_limit: dropped\n"));
```



```
2966         return (B_TRUE);
2967 }

2969 /*
2970  * Check if it is ok to send an IPv4 ICMP error packet in
2971  * response to the IPv4 packet in mp.
2972  * Free the message and return null if no
2973  * ICMP error packet should be sent.
2974  */
2975 static mblk_t *
2976 icmp_pkt_err_ok(mblk_t *mp, ip_recv_attr_t *ira)
2977 {
2978         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
2979         icmph_t *icmph;
2980         ipha_t  *ipha;
2981         uint_t  len_needed;

2983         if (!mp)
2984                 return (NULL);
2985         ipha = (ipha_t *)mp->b_rptr;
2986         if (ip_csum_hdr(ipha)) {
2987                 BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsInCksumErrs);
2988                 ip_drop_input("ipIfStatsInCksumErrs", mp, NULL);
2989                 freemsg(mp);
2990                 return (NULL);
2991         }
2992         if (ip_type_v4(ipha->ipha_dst, ipst) == IRE_BROADCAST ||
2993             ip_type_v4(ipha->ipha_src, ipst) == IRE_BROADCAST ||
2994             CLASSD(ipha->ipha_dst) ||
2995             CLASSD(ipha->ipha_src) ||
2996             (ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_OFFSET)) {
2997                 /* Note: only errors to the fragment with offset 0 */
2998                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
2999                 freemsg(mp);
3000                 return (NULL);
3001         }
3002         if (ipha->ipha_protocol == IPPROTO_ICMP) {
3003                 /*
3004                  * Check the ICMP type.  RFC 1122 sez:  don't send ICMP
3005                  * errors in response to any ICMP errors.
3006                  */
3007                 len_needed = IPH_HDR_LENGTH(ipha) + ICMPH_SIZE;
3008                 if (mp->b_wptr - mp->b_rptr < len_needed) {
3009                         if (!pullupmsg(mp, len_needed)) {
3010                                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
3011                                 freemsg(mp);
3012                                 return (NULL);
3013                         }
3014                         ipha = (ipha_t *)mp->b_rptr;
3015                 }
3016                 icmph = (icmph_t *)
3017                     (&((char *)ipha)[IPH_HDR_LENGTH(ipha)]);
3018                 switch (icmph->icmph_type) {
3019                 case ICMP_DEST_UNREACHABLE:
3020                 case ICMP_SOURCE_QUENCH:
3021                 case ICMP_TIME_EXCEEDED:
3022                 case ICMP_PARAM_PROBLEM:
3023                 case ICMP_REDIRECT:
3024                         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
3025                         freemsg(mp);
3026                         return (NULL);
3027                 default:
3028                         break;
3029                 }
3030         }
3031         /*
```

```
3032          * If this is a labeled system, then check to see if we're allowed to
3033          * send a response to this particular sender.  If not, then just drop.
3034          */
3035         if (is_system_labeled() && !tsol_can_reply_error(mp, ira)) {
3036                 ip2dbg(("icmp_pkt_err_ok: can't respond to packet\n"));
3037                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
3038                 freemsg(mp);
3039                 return (NULL);
3040         }
3041         if (icmp_err_rate_limit(ipst)) {
3042                 /*
3043                  * Only send ICMP error packets every so often.
3044                  * This should be done on a per port/source basis,
3045                  * but for now this will suffice.
3046                  */
3047                 freemsg(mp);
3048                 return (NULL);
3049         }
3050         return (mp);
3051 }

3053 /*
3054  * Called when a packet was sent out the same link that it arrived on.
3055  * Check if it is ok to send a redirect and then send it.
3056  */
3057 void
3058 ip_send_potential_redirect_v4(mblk_t *mp, ipha_t *ipha, ire_t *ire,
3059     ip_recv_attr_t *ira)
3060 {
3061         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
3062         ipaddr_t        src, nhop;
3063         mblk_t          *mp1;
3064         ire_t           *nhop_ire;

3066         /*
3067          * Check the source address to see if it originated
3068          * on the same logical subnet it is going back out on.
3069          * If so, we should be able to send it a redirect.
3070          * Avoid sending a redirect if the destination
3071          * is directly connected (i.e., we matched an IRE_ONLINK),
3072          * or if the packet was source routed out this interface.
3073          *
3074          * We avoid sending a redirect if the
3075          * destination is directly connected
3076          * because it is possible that multiple
3077          * IP subnets may have been configured on
3078          * the link, and the source may not
3079          * be on the same subnet as ip destination,
3080          * even though they are on the same
3081          * physical link.
3082          */
3083         if ((ire->ire_type & IRE_ONLINK) ||
3084             ip_source_routed(ipha, ipst))
3085                 return;

3087         nhop_ire = ire_nexthop(ire);
3088         if (nhop_ire == NULL)
3089                 return;

3091         nhop = nhop_ire->ire_addr;

3093         if (nhop_ire->ire_type & IRE_IF_CLONE) {
3094                 ire_t   *ire2;

3096                 /* Follow ire_dep_parent to find non-clone IRE_INTERFACE */
3097                 mutex_enter(&nhop_ire->ire_lock);
```



```
3098                 ire2 = nhop_ire->ire_dep_parent;
3099                 if (ire2 != NULL)
3100                         ire_refhold(ire2);
3101                 mutex_exit(&nhop_ire->ire_lock);
3102                 ire_refrele(nhop_ire);
3103                 nhop_ire = ire2;
3104         }
3105         if (nhop_ire == NULL)
3106                 return;

3108         ASSERT(!(nhop_ire->ire_type & IRE_IF_CLONE));

3110         src = ipha->ipha_src;

3112         /*
3113          * We look at the interface ire for the nexthop,
3114          * to see if ipha_src is in the same subnet
3115          * as the nexthop.
3116          */
3117         if ((src & nhop_ire->ire_mask) == (nhop & nhop_ire->ire_mask)) {
3118                 /*
3119                  * The source is directly connected.
3120                  */
3121                 mp1 = copymsg(mp);
3122                 if (mp1 != NULL) {
3123                         icmp_send_redirect(mp1, nhop, ira);
3124                 }
3125         }
3126         ire_refrele(nhop_ire);
3127 }

3129 /*
3130  * Generate an ICMP redirect message.
3131  */
3132 static void
3133 icmp_send_redirect(mblk_t *mp, ipaddr_t gateway, ip_recv_attr_t *ira)
3134 {
3135         icmph_t icmph;
3136         ip_stack_t *ipst = ira->ira_ill->ill_ipst;

3138         mp = icmp_pkt_err_ok(mp, ira);
3139         if (mp == NULL)
3140                 return;

3142         bzero(&icmph, sizeof (icmph_t));
3143         icmph.icmph_type = ICMP_REDIRECT;
3144         icmph.icmph_code = 1;
3145         icmph.icmph_rd_gateway = gateway;
3146         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutRedirects);
3147         icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
3148 }

3150 /*
3151  * Generate an ICMP time exceeded message.
3152  */
3153 void
3154 icmp_time_exceeded(mblk_t *mp, uint8_t code, ip_recv_attr_t *ira)
3155 {
3156         icmph_t icmph;
3157         ip_stack_t *ipst = ira->ira_ill->ill_ipst;

3159         mp = icmp_pkt_err_ok(mp, ira);
3160         if (mp == NULL)
3161                 return;

3163         bzero(&icmph, sizeof (icmph_t));
```

```
3164         icmph.icmph_type = ICMP_TIME_EXCEEDED;
3165         icmph.icmph_code = code;
3166         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutTimeExcds);
3167         icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
3168 }

3170 /*
3171  * Generate an ICMP unreachable message.
3172  * When called from ip_output side a minimal ip_recv_attr_t needs to be
3173  * constructed by the caller.
3174  */
3175 void
3176 icmp_unreachable(mblk_t *mp, uint8_t code, ip_recv_attr_t *ira)
3177 {
3178         icmph_t icmph;
3179         ip_stack_t *ipst = ira->ira_ill->ill_ipst;

3181         mp = icmp_pkt_err_ok(mp, ira);
3182         if (mp == NULL)
3183                 return;

3185         bzero(&icmph, sizeof (icmph_t));
3186         icmph.icmph_type = ICMP_DEST_UNREACHABLE;
3187         icmph.icmph_code = code;
3188         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDestUnreachs);
3189         icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
3190 }

3192 /*
3193  * Latch in the IPsec state for a stream based the policy in the listener
3194  * and the actions in the ip_recv_attr_t.
3195  * Called directly from TCP and SCTP.
3196  */
3197 boolean_t
3198 ip_ipsec_policy_inherit(conn_t *connp, conn_t *lconnp, ip_recv_attr_t *ira)
3199 {
3200         ASSERT(lconnp->conn_policy != NULL);
3201         ASSERT(connp->conn_policy == NULL);

3203         IPPH_REFHOLD(lconnp->conn_policy);
3204         connp->conn_policy = lconnp->conn_policy;

3206         if (ira->ira_ipsec_action != NULL) {
3207                 if (connp->conn_latch == NULL) {
3208                         connp->conn_latch = iplatch_create();
3209                         if (connp->conn_latch == NULL)
3210                                 return (B_FALSE);
3211                 }
3212                 ipsec_latch_inbound(connp, ira);
3213         }
3214         return (B_TRUE);
3215 }

3217 /*
3218  * Verify whether or not the IP address is a valid local address.
3219  * Could be a unicast, including one for a down interface.
3220  * If allow_mcbc then a multicast or broadcast address is also
3221  * acceptable.
3222  *
3223  * In the case of a broadcast/multicast address, however, the
3224  * upper protocol is expected to reset the src address
3225  * to zero when we return IPVL_MCAST/IPVL_BCAST so that
3226  * no packets are emitted with broadcast/multicast address as
3227  * source address (that violates hosts requirements RFC 1122)
3228  * The addresses valid for bind are:
3229  *      (1) - INADDR_ANY (0)
```



```
3230  *      (2) - IP address of an UP interface
3231  *      (3) - IP address of a DOWN interface
3232  *      (4) - valid local IP broadcast addresses. In this case
3233  *      the conn will only receive packets destined to
3234  *      the specified broadcast address.
3235  *      (5) - a multicast address. In this case
3236  *      the conn will only receive packets destined to
3237  *      the specified multicast address. Note: the
3238  *      application still has to issue an
3239  *      IP_ADD_MEMBERSHIP socket option.
3240  *
3241  * In all the above cases, the bound address must be valid in the current zone.
3242  * When the address is loopback, multicast or broadcast, there might be many
3243  * matching IREs so bind has to look up based on the zone.
3244  */
3245 ip_laddr_t
3246 ip_laddr_verify_v4(ipaddr_t src_addr, zoneid_t zoneid,
3247     ip_stack_t *ipst, boolean_t allow_mcbc)
3248 {
3249         ire_t *src_ire;

3251         ASSERT(src_addr != INADDR_ANY);

3253         src_ire = ire_ftable_lookup_v4(src_addr, 0, 0, 0,
3254             NULL, zoneid, NULL, MATCH_IRE_ZONEONLY, 0, ipst, NULL);

3256         /*
3257          * If an address other than in6addr_any is requested,
3258          * we verify that it is a valid address for bind
3259          * Note: Following code is in if-else-if form for
3260          * readability compared to a condition check.
3261          */
3262         if (src_ire != NULL && (src_ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK))) {
3263                 /*
3264                  * (2) Bind to address of local UP interface
3265                  */
3266                 ire_refrele(src_ire);
3267                 return (IPVL_UNICAST_UP);
3268         } else if (src_ire != NULL && src_ire->ire_type & IRE_BROADCAST) {
3269                 /*
3270                  * (4) Bind to broadcast address
3271                  */
3272                 ire_refrele(src_ire);
3273                 if (allow_mcbc)
3274                         return (IPVL_BCAST);
3275                 else
3276                         return (IPVL_BAD);
3277         } else if (CLASSD(src_addr)) {
3278                 /* (5) bind to multicast address. */
3279                 if (src_ire != NULL)
3280                         ire_refrele(src_ire);

3282                 if (allow_mcbc)
3283                         return (IPVL_MCAST);
3284                 else
3285                         return (IPVL_BAD);
3286         } else {
3287                 ipif_t *ipif;

3289                 /*
3290                  * (3) Bind to address of local DOWN interface?
3291                  * (ipif_lookup_addr() looks up all interfaces
3292                  * but we do not get here for UP interfaces
3293                  * - case (2) above)
3294                  */
3295                 if (src_ire != NULL)
```

```
3296                        ire_refrele(src_ire);

3298                ipif = ipif_lookup_addr(src_addr, NULL, zoneid, ipst);
3299                if (ipif == NULL)
3300                        return (IPVL_BAD);

3302                /* Not a useful source? */
3303                if (ipif->ipif_flags & (IPIF_NOLOCAL | IPIF_ANYCAST)) {
3304                        ipif_refrele(ipif);
3305                        return (IPVL_BAD);
3306                }
3307                ipif_refrele(ipif);
3308                return (IPVL_UNICAST_DOWN);
3309        }
3310 }

3312 /*
3313  * Insert in the bind fanout for IPv4 and IPv6.
3314  * The caller should already have used ip_laddr_verify_v*() before calling
3315  * this.
3316  */
3317 int
3318 ip_laddr_fanout_insert(conn_t *connp)
3319 {
3320        int             error;

3322        /*
3323         * Allow setting new policies. For example, disconnects result
3324         * in us being called. As we would have set conn_policy_cached
3325         * to B_TRUE before, we should set it to B_FALSE, so that policy
3326         * can change after the disconnect.
3327         */
3328        connp->conn_policy_cached = B_FALSE;

3330        error = ipcl_bind_insert(connp);
3331        if (error != 0) {
3332                if (connp->conn_anon_port) {
3333                        (void) tsol_mlp_anon(crgetzone(connp->conn_cred),
3334                            connp->conn_mlp_type, connp->conn_proto,
3335                            ntohs(connp->conn_lport), B_FALSE);
3336                }
3337                connp->conn_mlp_type = mlptSingle;
3338        }
3339        return (error);
3340 }

3342 /*
3343  * Verify that both the source and destination addresses are valid. If
3344  * IPDF_VERIFY_DST is not set, then the destination address may be unreachable,
3345  * i.e. have no route to it.  Protocols like TCP want to verify destination
3346  * reachability, while tunnels do not.
3347  *
3348  * Determine the route, the interface, and (optionally) the source address
3349  * to use to reach a given destination.
3350  * Note that we allow connect to broadcast and multicast addresses when
3351  * IPDF_ALLOW_MCBC is set.
3352  * first_hop and dst_addr are normally the same, but if source routing
3353  * they will differ; in that case the first_hop is what we'll use for the
3354  * routing lookup but the dce and label checks will be done on dst_addr,
3355  *
3356  * If uinfo is set, then we fill in the best available information
3357  * we have for the destination. This is based on (in priority order) any
3358  * metrics and path MTU stored in a dce_t, route metrics, and finally the
3359  * ill_mtu/ill_mc_mtu.
3360  *
3361  * Tsol note: If we have a source route then dst_addr != firsthop. But we
```



```
3362  * always do the label check on dst_addr.
3363  */
3364 int
3365 ip_set_destination_v4(ipaddr_t *src_addrp, ipaddr_t dst_addr, ipaddr_t firsthop,
3366     ip_xmit_attr_t *ixa, iulp_t *uinfo, uint32_t flags, uint_t mac_mode)
3367 {
3368        ire_t           *ire = NULL;
3369        int             error = 0;
3370        ipaddr_t        setsrc;                         /* RTF_SETSRC */
3371        zoneid_t        zoneid = ixa->ixa_zoneid;       /* Honors SO_ALLZONES */
3372        ip_stack_t      *ipst = ixa->ixa_ipst;
3373        dce_t           *dce;
3374        uint_t          pmtu;
3375        uint_t          generation;
3376        nce_t           *nce;
3377        ill_t           *ill = NULL;
3378        boolean_t       multirt = B_FALSE;

3380        ASSERT(ixa->ixa_flags & IXAF_IS_IPV4);

3382        /*
3383         * We never send to zero; the ULPs map it to the loopback address.
3384         * We can't allow it since we use zero to mean unitialized in some
3385         * places.
3386         */
3387        ASSERT(dst_addr != INADDR_ANY);

3389        if (is_system_labeled()) {
3390                ts_label_t *tsl = NULL;

3392                error = tsol_check_dest(ixa->ixa_tsl, &dst_addr, IPV4_VERSION,
3393                    mac_mode, (flags & IPDF_ZONE_IS_GLOBAL) != 0, &tsl);
3394                if (error != 0)
3395                        return (error);
3396                if (tsl != NULL) {
3397                        /* Update the label */
3398                        ip_xmit_attr_replace_tsl(ixa, tsl);
3399                }
3400        }

3402        setsrc = INADDR_ANY;
3403        /*
3404         * Select a route; For IPMP interfaces, we would only select
3405         * a "hidden" route (i.e., going through a specific under_ill)
3406         * if ixa_ifindex has been specified.
3407         */
3408        ire = ip_select_route_v4(firsthop, *src_addrp, ixa,
3409            &generation, &setsrc, &error, &multirt);
3410        ASSERT(ire != NULL);    /* IRE_NOROUTE if none found */
3411        if (error != 0)
3412                goto bad_addr;

3414        /*
3415         * ire can't be a broadcast or multicast unless IPDF_ALLOW_MCBC is set.
3416         * If IPDF_VERIFY_DST is set, the destination must be reachable;
3417         * Otherwise the destination needn't be reachable.
3418         *
3419         * If we match on a reject or black hole, then we've got a
3420         * local failure.  May as well fail out the connect() attempt,
3421         * since it's never going to succeed.
3422         */
3423        if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) {
3424                /*
3425                 * If we're verifying destination reachability, we always want
3426                 * to complain here.
3427                 *
```

```
3428                  * If we're not verifying destination reachability but the
3429                  * destination has a route, we still want to fail on the
3430                  * temporary address and broadcast address tests.
3431                  *
3432                  * In both cases do we let the code continue so some reasonable
3433                  * information is returned to the caller. That enables the
3434                  * caller to use (and even cache) the IRE. conn_ip_ouput will
3435                  * use the generation mismatch path to check for the unreachable
3436                  * case thereby avoiding any specific check in the main path.
3437                  */
3438                 ASSERT(generation == IRE_GENERATION_VERIFY);
3439                 if (flags & IPDF_VERIFY_DST) {
3440                         /*
3441                          * Set errno but continue to set up ixa_ire to be
3442                          * the RTF_REJECT|RTF_BLACKHOLE IRE.
3443                          * That allows callers to use ip_output to get an
3444                          * ICMP error back.
3445                          */
3446                         if (!(ire->ire_type & IRE_HOST))
3447                                 error = ENETUNREACH;
3448                         else
3449                                 error = EHOSTUNREACH;
3450                 }
3451         }

3453         if ((ire->ire_type & (IRE_BROADCAST|IRE_MULTICAST)) &&
3454             !(flags & IPDF_ALLOW_MCBC)) {
3455                 ire_refrele(ire);
3456                 ire = ire_reject(ipst, B_FALSE);
3457                 generation = IRE_GENERATION_VERIFY;
3458                 error = ENETUNREACH;
3459         }

3461         /* Cache things */
3462         if (ixa->ixa_ire != NULL)
3463                 ire_refrele_notr(ixa->ixa_ire);
3464 #ifdef DEBUG
3465         ire_refhold_notr(ire);
3466         ire_refrele(ire);
3467 #endif
3468         ixa->ixa_ire = ire;
3469         ixa->ixa_ire_generation = generation;

3471         /*
3472          * Ensure that ixa_dce is always set any time that ixa_ire is set,
3473          * since some callers will send a packet to conn_ip_output() even if
3474          * there's an error.
3475          */
3476         if (flags & IPDF_UNIQUE_DCE) {
3477                 /* Fallback to the default dce if allocation fails */
3478                 dce = dce_lookup_and_add_v4(dst_addr, ipst);
3479                 if (dce != NULL)
3480                         generation = dce->dce_generation;
3481                 else
3482                         dce = dce_lookup_v4(dst_addr, ipst, &generation);
3483         } else {
3484                 dce = dce_lookup_v4(dst_addr, ipst, &generation);
3485         }
3486         ASSERT(dce != NULL);
3487         if (ixa->ixa_dce != NULL)
3488                 dce_refrele_notr(ixa->ixa_dce);
3489 #ifdef DEBUG
3490         dce_refhold_notr(dce);
3491         dce_refrele(dce);
3492 #endif
3493         ixa->ixa_dce = dce;
```



```
3494         ixa->ixa_dce_generation = generation;

3496         /*
3497          * For multicast with multirt we have a flag passed back from
3498          * ire_lookup_multi_ill_v4 since we don't have an IRE for each
3499          * possible multicast address.
3500          * We also need a flag for multicast since we can't check
3501          * whether RTF_MULTIRT is set in ixa_ire for multicast.
3502          */
3503         if (multirt) {
3504                 ixa->ixa_postfragfn = ip_postfrag_multirt_v4;
3505                 ixa->ixa_flags |= IXAF_MULTIRT_MULTICAST;
3506         } else {
3507                 ixa->ixa_postfragfn = ire->ire_postfragfn;
3508                 ixa->ixa_flags &= ~IXAF_MULTIRT_MULTICAST;
3509         }
3510         if (!(ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))) {
3511                 /* Get an nce to cache. */
3512                 nce = ire_to_nce(ire, firsthop, NULL);
3513                 if (nce == NULL) {
3514                         /* Allocation failure? */
3515                         ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
3516                 } else {
3517                         if (ixa->ixa_nce != NULL)
3518                                 nce_refrele(ixa->ixa_nce);
3519                         ixa->ixa_nce = nce;
3520                 }
3521         }

3523         /*
3524          * If the source address is a loopback address, the
3525          * destination had best be local or multicast.
3526          * If we are sending to an IRE_LOCAL using a loopback source then
3527          * it had better be the same zoneid.
3528          */
3529         if (*src_addrp == htonl(INADDR_LOOPBACK)) {
3530                 if ((ire->ire_type & IRE_LOCAL) && ire->ire_zoneid != zoneid) {
3531                         ire = NULL;     /* Stored in ixa_ire */
3532                         error = EADDRNOTAVAIL;
3533                         goto bad_addr;
3534                 }
3535                 if (!(ire->ire_type & (IRE_LOOPBACK|IRE_LOCAL|IRE_MULTICAST))) {
3536                         ire = NULL;     /* Stored in ixa_ire */
3537                         error = EADDRNOTAVAIL;
3538                         goto bad_addr;
3539                 }
3540         }
3541         if (ire->ire_type & IRE_BROADCAST) {
3542                 /*
3543                  * If the ULP didn't have a specified source, then we
3544                  * make sure we reselect the source when sending
3545                  * broadcasts out different interfaces.
3546                  */
3547                 if (flags & IPDF_SELECT_SRC)
3548                         ixa->ixa_flags |= IXAF_SET_SOURCE;
3549                 else
3550                         ixa->ixa_flags &= ~IXAF_SET_SOURCE;
3551         }

3553         /*
3554          * Does the caller want us to pick a source address?
3555          */
3556         if (flags & IPDF_SELECT_SRC) {
3557                 ipaddr_t        src_addr;

3559                 /*
```

```
3560                 * We use use ire_nexthop_ill to avoid the under ipmp
3561                 * interface for source address selection. Note that for ipmp
3562                 * probe packets, ixa_ifindex would have been specified, and
3563                 * the ip_select_route() invocation would have picked an ire
3564                 * will ire_ill pointing at an under interface.
3565                 */
3566                ill = ire_nexthop_ill(ire);

3568                /* If unreachable we have no ill but need some source */
3569                if (ill == NULL) {
3570                        src_addr = htonl(INADDR_LOOPBACK);
3571                        /* Make sure we look for a better source address */
3572                        generation = SRC_GENERATION_VERIFY;
3573                } else {
3574                        error = ip_select_source_v4(ill, setsrc, dst_addr,
3575                            ixa->ixa_multicast_ifaddr, zoneid,
3576                            ipst, &src_addr, &generation, NULL);
3577                        if (error != 0) {
3578                                ire = NULL;     /* Stored in ixa_ire */
3579                                goto bad_addr;
3580                        }
3581                }

3583                /*
3584                 * We allow the source address to to down.
3585                 * However, we check that we don't use the loopback address
3586                 * as a source when sending out on the wire.
3587                 */
3588                if ((src_addr == htonl(INADDR_LOOPBACK)) &&
3589                    !(ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK|IRE_MULTICAST)) &&
3590                    !(ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))) {
3591                        ire = NULL;     /* Stored in ixa_ire */
3592                        error = EADDRNOTAVAIL;
3593                        goto bad_addr;
3594                }

3596                *src_addrp = src_addr;
3597                ixa->ixa_src_generation = generation;
3598        }

3600        /*
3601         * Make sure we don't leave an unreachable ixa_nce in place
3602         * since ip_select_route is used when we unplumb i.e., remove
3603         * references on ixa_ire, ixa_nce, and ixa_dce.
3604         */
3605        nce = ixa->ixa_nce;
3606        if (nce != NULL && nce->nce_is_condemned) {
3607                nce_refrele(nce);
3608                ixa->ixa_nce = NULL;
3609                ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
3610        }

3612        /*
3613         * The caller has set IXAF_PMTU_DISCOVERY if path MTU is desired.
3614         * However, we can't do it for IPv4 multicast or broadcast.
3615         */
3616        if (ire->ire_type & (IRE_BROADCAST|IRE_MULTICAST))
3617                ixa->ixa_flags &= ~IXAF_PMTU_DISCOVERY;

3619        /*
3620         * Set initial value for fragmentation limit. Either conn_ip_output
3621         * or ULP might updates it when there are routing changes.
3622         * Handles a NULL ixa_ire->ire_ill or a NULL ixa_nce for RTF_REJECT.
3623         */
3624        pmtu = ip_get_pmtu(ixa);
3625        ixa->ixa_fragsize = pmtu;
```



```
3626        /* Make sure ixa_fragsize and ixa_pmtu remain identical */
3627        if (ixa->ixa_flags & IXAF_VERIFY_PMTU)
3628                ixa->ixa_pmtu = pmtu;

3630        /*
3631         * Extract information useful for some transports.
3632         * First we look for DCE metrics. Then we take what we have in
3633         * the metrics in the route, where the offlink is used if we have
3634         * one.
3635         */
3636        if (uinfo != NULL) {
3637                bzero(uinfo, sizeof (*uinfo));

3639                if (dce->dce_flags & DCEF_UINFO)
3640                        *uinfo = dce->dce_uinfo;

3642                rts_merge_metrics(uinfo, &ire->ire_metrics);

3644                /* Allow ire_metrics to decrease the path MTU from above */
3645                if (uinfo->iulp_mtu == 0 || uinfo->iulp_mtu > pmtu)
3646                        uinfo->iulp_mtu = pmtu;

3648                uinfo->iulp_localnet = (ire->ire_type & IRE_ONLINK) != 0;
3649                uinfo->iulp_loopback = (ire->ire_type & IRE_LOOPBACK) != 0;
3650                uinfo->iulp_local = (ire->ire_type & IRE_LOCAL) != 0;
3651        }

3653        if (ill != NULL)
3654                ill_refrele(ill);

3656        return (error);

3658 bad_addr:
3659        if (ire != NULL)
3660                ire_refrele(ire);

3662        if (ill != NULL)
3663                ill_refrele(ill);

3665        /*
3666         * Make sure we don't leave an unreachable ixa_nce in place
3667         * since ip_select_route is used when we unplumb i.e., remove
3668         * references on ixa_ire, ixa_nce, and ixa_dce.
3669         */
3670        nce = ixa->ixa_nce;
3671        if (nce != NULL && nce->nce_is_condemned) {
3672                nce_refrele(nce);
3673                ixa->ixa_nce = NULL;
3674                ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
3675        }

3677        return (error);
3678 }


3681 /*
3682  * Get the base MTU for the case when path MTU discovery is not used.
3683  * Takes the MTU of the IRE into account.
3684  */
3685 uint_t
3686 ip_get_base_mtu(ill_t *ill, ire_t *ire)
3687 {
3688        uint_t mtu;
3689        uint_t iremtu = ire->ire_metrics.iulp_mtu;

3691        if (ire->ire_type & (IRE_MULTICAST|IRE_BROADCAST))
```

```
3692                mtu = ill->ill_mc_mtu;
3693        else
3694                mtu = ill->ill_mtu;

3696        if (iremtu != 0 && iremtu < mtu)
3697                mtu = iremtu;

3699        return (mtu);
3700 }

3702 /*
3703  * Get the PMTU for the attributes. Handles both IPv4 and IPv6.
3704  * Assumes that ixa_ire, dce, and nce have already been set up.
3705  *
3706  * The caller has set IXAF_PMTU_DISCOVERY if path MTU discovery is desired.
3707  * We avoid path MTU discovery if it is disabled with ndd.
3708  * Furtermore, if the path MTU is too small, then we don't set DF for IPv4.
3709  *
3710  * NOTE: We also used to turn it off for source routed packets. That
3711  * is no longer required since the dce is per final destination.
3712  */
3713 uint_t
3714 ip_get_pmtu(ip_xmit_attr_t *ixa)
3715 {
3716        ip_stack_t      *ipst = ixa->ixa_ipst;
3717        dce_t           *dce;
3718        nce_t           *nce;
3719        ire_t           *ire;
3720        uint_t          pmtu;

3722        ire = ixa->ixa_ire;
3723        dce = ixa->ixa_dce;
3724        nce = ixa->ixa_nce;

3726        /*
3727         * If path MTU discovery has been turned off by ndd, then we ignore
3728         * any dce_pmtu and for IPv4 we will not set DF.
3729         */
3730        if (!ipst->ips_ip_path_mtu_discovery)
3731                ixa->ixa_flags &= ~IXAF_PMTU_DISCOVERY;

3733        pmtu = IP_MAXPACKET;
3734        /*
3735         * Decide whether whether IPv4 sets DF
3736         * For IPv6 "no DF" means to use the 1280 mtu
3737         */
3738        if (ixa->ixa_flags & IXAF_PMTU_DISCOVERY) {
3739                ixa->ixa_flags |= IXAF_PMTU_IPV4_DF;
3740        } else {
3741                ixa->ixa_flags &= ~IXAF_PMTU_IPV4_DF;
3742                if (!(ixa->ixa_flags & IXAF_IS_IPV4))
3743                        pmtu = IPV6_MIN_MTU;
3744        }

3746        /* Check if the PMTU is to old before we use it */
3747        if ((dce->dce_flags & DCEF_PMTU) &&
3748            TICK_TO_SEC(ddi_get_lbolt64()) - dce->dce_last_change_time >
3749            ipst->ips_ip_pathmtu_interval) {
3750                /*
3751                 * Older than 20 minutes. Drop the path MTU information.
3752                 */
3753                mutex_enter(&dce->dce_lock);
3754                dce->dce_flags &= ~(DCEF_PMTU|DCEF_TOO_SMALL_PMTU);
3755                dce->dce_last_change_time = TICK_TO_SEC(ddi_get_lbolt64());
3756                mutex_exit(&dce->dce_lock);
3757                dce_increment_generation(dce);
```



```
3758        }

3760        /* The metrics on the route can lower the path MTU */
3761        if (ire->ire_metrics.iulp_mtu != 0 &&
3762            ire->ire_metrics.iulp_mtu < pmtu)
3763                pmtu = ire->ire_metrics.iulp_mtu;

3765        /*
3766         * If the path MTU is smaller than some minimum, we still use dce_pmtu
3767         * above (would be 576 for IPv4 and 1280 for IPv6), but we clear
3768         * IXAF_PMTU_IPV4_DF so that we avoid setting DF for IPv4.
3769         */
3770        if (ixa->ixa_flags & IXAF_PMTU_DISCOVERY) {
3771                if (dce->dce_flags & DCEF_PMTU) {
3772                        if (dce->dce_pmtu < pmtu)
3773                                pmtu = dce->dce_pmtu;

3775                        if (dce->dce_flags & DCEF_TOO_SMALL_PMTU) {
3776                                ixa->ixa_flags |= IXAF_PMTU_TOO_SMALL;
3777                                ixa->ixa_flags &= ~IXAF_PMTU_IPV4_DF;
3778                        } else {
3779                                ixa->ixa_flags &= ~IXAF_PMTU_TOO_SMALL;
3780                                ixa->ixa_flags |= IXAF_PMTU_IPV4_DF;
3781                        }
3782                } else {
3783                        ixa->ixa_flags &= ~IXAF_PMTU_TOO_SMALL;
3784                        ixa->ixa_flags |= IXAF_PMTU_IPV4_DF;
3785                }
3786        }

3788        /*
3789         * If we have an IRE_LOCAL we use the loopback mtu instead of
3790         * the ill for going out the wire i.e., IRE_LOCAL gets the same
3791         * mtu as IRE_LOOPBACK.
3792         */
3793        if (ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK)) {
3794                uint_t loopback_mtu;

3796                loopback_mtu = (ire->ire_ipversion == IPV6_VERSION) ?
3797                    ip_loopback_mtu_v6plus : ip_loopback_mtuplus;

3799                if (loopback_mtu < pmtu)
3800                        pmtu = loopback_mtu;
3801        } else if (nce != NULL) {
3802                /*
3803                 * Make sure we don't exceed the interface MTU.
3804                 * In the case of RTF_REJECT or RTF_BLACKHOLE we might not have
3805                 * an ill. We'd use the above IP_MAXPACKET in that case just
3806                 * to tell the transport something larger than zero.
3807                 */
3808                if (ire->ire_type & (IRE_MULTICAST|IRE_BROADCAST)) {
3809                        if (nce->nce_common->ncec_ill->ill_mc_mtu < pmtu)
3810                                pmtu = nce->nce_common->ncec_ill->ill_mc_mtu;
3811                        if (nce->nce_common->ncec_ill != nce->nce_ill &&
3812                            nce->nce_ill->ill_mc_mtu < pmtu) {
3813                                /*
3814                                 * for interfaces in an IPMP group, the mtu of
3815                                 * the nce_ill (under_ill) could be different
3816                                 * from the mtu of the ncec_ill, so we take the
3817                                 * min of the two.
3818                                 */
3819                                pmtu = nce->nce_ill->ill_mc_mtu;
3820                        }
3821                } else {
3822                        if (nce->nce_common->ncec_ill->ill_mtu < pmtu)
3823                                pmtu = nce->nce_common->ncec_ill->ill_mtu;
```

```
3824                        if (nce->nce_common->ncec_ill != nce->nce_ill &&
3825                            nce->nce_ill->ill_mtu < pmtu) {
3826                                /*
3827                                 * for interfaces in an IPMP group, the mtu of
3828                                 * the nce_ill (under_ill) could be different
3829                                 * from the mtu of the ncec_ill, so we take the
3830                                 * min of the two.
3831                                 */
3832                                pmtu = nce->nce_ill->ill_mtu;
3833                        }
3834                }
3835        }

3837        /*
3838         * Handle the IPV6_USE_MIN_MTU socket option or ancillary data.
3839         * Only applies to IPv6.
3840         */
3841        if (!(ixa->ixa_flags & IXAF_IS_IPV4)) {
3842                if (ixa->ixa_flags & IXAF_USE_MIN_MTU) {
3843                        switch (ixa->ixa_use_min_mtu) {
3844                        case IPV6_USE_MIN_MTU_MULTICAST:
3845                                if (ire->ire_type & IRE_MULTICAST)
3846                                        pmtu = IPV6_MIN_MTU;
3847                                break;
3848                        case IPV6_USE_MIN_MTU_ALWAYS:
3849                                pmtu = IPV6_MIN_MTU;
3850                                break;
3851                        case IPV6_USE_MIN_MTU_NEVER:
3852                                break;
3853                        }
3854                } else {
3855                        /* Default is IPV6_USE_MIN_MTU_MULTICAST */
3856                        if (ire->ire_type & IRE_MULTICAST)
3857                                pmtu = IPV6_MIN_MTU;
3858                }
3859        }

3861        /*
3862         * After receiving an ICMPv6 "packet too big" message with a
3863         * MTU < 1280, and for multirouted IPv6 packets, the IP layer
3864         * will insert a 8-byte fragment header in every packet. We compensate
3865         * for those cases by returning a smaller path MTU to the ULP.
3866         *
3867         * In the case of CGTP then ip_output will add a fragment header.
3868         * Make sure there is room for it by telling a smaller number
3869         * to the transport.
3870         *
3871         * When IXAF_IPV6_ADDR_FRAGHDR we subtract the frag hdr here
3872         * so the ULPs consistently see a iulp_pmtu and ip_get_pmtu()
3873         * which is the size of the packets it can send.
3874         */
3875        if (!(ixa->ixa_flags & IXAF_IS_IPV4)) {
3876                if ((dce->dce_flags & DCEF_TOO_SMALL_PMTU) ||
3877                    (ire->ire_flags & RTF_MULTIRT) ||
3878                    (ixa->ixa_flags & IXAF_MULTIRT_MULTICAST)) {
3879                        pmtu -= sizeof (ip6_frag_t);
3880                        ixa->ixa_flags |= IXAF_IPV6_ADD_FRAGHDR;
3881                }
3882        }

3884        return (pmtu);
3885 }

3887 /*
3888  * Carve "len" bytes out of an mblk chain, consuming any we empty, and duping
3889  * the final piece where we don't.  Return a pointer to the first mblk in the
```



```
3890  * result, and update the pointer to the next mblk to chew on.  If anything
3891  * goes wrong (i.e., dupb fails), we waste everything in sight and return a
3892  * NULL pointer.
3893  */
3894 mblk_t *
3895 ip_carve_mp(mblk_t **mpp, ssize_t len)
3896 {
3897         mblk_t  *mp0;
3898         mblk_t  *mp1;
3899         mblk_t  *mp2;

3901         if (!len || !mpp || !(mp0 = *mpp))
3902                 return (NULL);
3903         /* If we aren't going to consume the first mblk, we need a dup. */
3904         if (mp0->b_wptr - mp0->b_rptr > len) {
3905                 mp1 = dupb(mp0);
3906                 if (mp1) {
3907                         /* Partition the data between the two mblks. */
3908                         mp1->b_wptr = mp1->b_rptr + len;
3909                         mp0->b_rptr = mp1->b_wptr;
3910                         /*
3911                          * after adjustments if mblk not consumed is now
3912                          * unaligned, try to align it. If this fails free
3913                          * all messages and let upper layer recover.
3914                          */
3915                         if (!OK_32PTR(mp0->b_rptr)) {
3916                                 if (!pullupmsg(mp0, -1)) {
3917                                         freemsg(mp0);
3918                                         freemsg(mp1);
3919                                         *mpp = NULL;
3920                                         return (NULL);
3921                                 }
3922                         }
3923                 }
3924                 return (mp1);
3925         }
3926         /* Eat through as many mblks as we need to get len bytes. */
3927         len -= mp0->b_wptr - mp0->b_rptr;
3928         for (mp2 = mp1 = mp0; (mp2 = mp2->b_cont) != 0 && len; mp1 = mp2) {
3929                 if (mp2->b_wptr - mp2->b_rptr > len) {
3930                         /*
3931                          * We won't consume the entire last mblk.  Like
3932                          * above, dup and partition it.
3933                          */
3934                         mp1->b_cont = dupb(mp2);
3935                         mp1 = mp1->b_cont;
3936                         if (!mp1) {
3937                                 /*
3938                                  * Trouble.  Rather than go to a lot of
3939                                  * trouble to clean up, we free the messages.
3940                                  * This won't be any worse than losing it on
3941                                  * the wire.
3942                                  */
3943                                 freemsg(mp0);
3944                                 freemsg(mp2);
3945                                 *mpp = NULL;
3946                                 return (NULL);
3947                         }
3948                         mp1->b_wptr = mp1->b_rptr + len;
3949                         mp2->b_rptr = mp1->b_wptr;
3950                         /*
3951                          * after adjustments if mblk not consumed is now
3952                          * unaligned, try to align it. If this fails free
3953                          * all messages and let upper layer recover.
3954                          */
3955                         if (!OK_32PTR(mp2->b_rptr)) {
```

```
3956                                if (!pullupmsg(mp2, -1)) {
3957                                        freemsg(mp0);
3958                                        freemsg(mp2);
3959                                        *mpp = NULL;
3960                                        return (NULL);
3961                                }
3962                        }
3963                        *mpp = mp2;
3964                        return (mp0);
3965                }
3966                /* Decrement len by the amount we just got. */
3967                len -= mp2->b_wptr - mp2->b_rptr;
3968        }
3969        /*
3970         * len should be reduced to zero now.  If not our caller has
3971         * screwed up.
3972         */
3973        if (len) {
3974                /* Shouldn't happen! */
3975                freemsg(mp0);
3976                *mpp = NULL;
3977                return (NULL);
3978        }
3979        /*
3980         * We consumed up to exactly the end of an mblk.  Detach the part
3981         * we are returning from the rest of the chain.
3982         */
3983        mp1->b_cont = NULL;
3984        *mpp = mp2;
3985        return (mp0);
3986 }

3988 /* The ill stream is being unplumbed. Called from ip_close */
3989 int
3990 ip_modclose(ill_t *ill)
3991 {
3992        boolean_t success;
3993        ipsq_t  *ipsq;
3994        ipif_t  *ipif;
3995        queue_t *q = ill->ill_rq;
3996        ip_stack_t      *ipst = ill->ill_ipst;
3997        int     i;
3998        arl_ill_common_t *ai = ill->ill_common;

4000        /*
4001         * The punlink prior to this may have initiated a capability
4002         * negotiation. But ipsq_enter will block until that finishes or
4003         * times out.
4004         */
4005        success = ipsq_enter(ill, B_FALSE, NEW_OP);

4007        /*
4008         * Open/close/push/pop is guaranteed to be single threaded
4009         * per stream by STREAMS. FS guarantees that all references
4010         * from top are gone before close is called. So there can't
4011         * be another close thread that has set CONDEMNED on this ill.
4012         * and cause ipsq_enter to return failure.
4013         */
4014        ASSERT(success);
4015        ipsq = ill->ill_phyint->phyint_ipsq;

4017        /*
4018         * Mark it condemned. No new reference will be made to this ill.
4019         * Lookup functions will return an error. Threads that try to
4020         * increment the refcnt must check for ILL_CAN_LOOKUP. This ensures
4021         * that the refcnt will drop down to zero.
```



```
4022         */
4023        mutex_enter(&ill->ill_lock);
4024        ill->ill_state_flags |= ILL_CONDEMNED;
4025        for (ipif = ill->ill_ipif; ipif != NULL;
4026            ipif = ipif->ipif_next) {
4027                ipif->ipif_state_flags |= IPIF_CONDEMNED;
4028        }
4029        /*
4030         * Wake up anybody waiting to enter the ipsq. ipsq_enter
4031         * returns  error if ILL_CONDEMNED is set
4032         */
4033        cv_broadcast(&ill->ill_cv);
4034        mutex_exit(&ill->ill_lock);

4036        /*
4037         * Send all the deferred DLPI messages downstream which came in
4038         * during the small window right before ipsq_enter(). We do this
4039         * without waiting for the ACKs because all the ACKs for M_PROTO
4040         * messages are ignored in ip_rput() when ILL_CONDEMNED is set.
4041         */
4042        ill_dlpi_send_deferred(ill);

4044        /*
4045         * Shut down fragmentation reassembly.
4046         * ill_frag_timer won't start a timer again.
4047         * Now cancel any existing timer
4048         */
4049        (void) untimeout(ill->ill_frag_timer_id);
4050        (void) ill_frag_timeout(ill, 0);

4052        /*
4053         * Call ill_delete to bring down the ipifs, ilms and ill on
4054         * this ill. Then wait for the refcnts to drop to zero.
4055         * ill_is_freeable checks whether the ill is really quiescent.
4056         * Then make sure that threads that are waiting to enter the
4057         * ipsq have seen the error returned by ipsq_enter and have
4058         * gone away. Then we call ill_delete_tail which does the
4059         * DL_UNBIND_REQ with the driver and then qprocsoff.
4060         */
4061        ill_delete(ill);
4062        mutex_enter(&ill->ill_lock);
4063        while (!ill_is_freeable(ill))
4064                cv_wait(&ill->ill_cv, &ill->ill_lock);

4066        while (ill->ill_waiters)
4067                cv_wait(&ill->ill_cv, &ill->ill_lock);

4069        mutex_exit(&ill->ill_lock);

4071        /*
4072         * ill_delete_tail drops reference on ill_ipst, but we need to keep
4073         * it held until the end of the function since the cleanup
4074         * below needs to be able to use the ip_stack_t.
4075         */
4076        netstack_hold(ipst->ips_netstack);

4078        /* qprocsoff is done via ill_delete_tail */
4079        ill_delete_tail(ill);
4080        /*
4081         * synchronously wait for arp stream to unbind. After this, we
4082         * cannot get any data packets up from the driver.
4083         */
4084        arp_unbind_complete(ill);
4085        ASSERT(ill->ill_ipst == NULL);

4087        /*
```

```
4088          * Walk through all conns and qenable those that have queued data.
4089          * Close synchronization needs this to
4090          * be done to ensure that all upper layers blocked
4091          * due to flow control to the closing device
4092          * get unblocked.
4093          */
4094         ip1dbg(("ip_wsrv: walking\n"));
4095         for (i = 0; i < TX_FANOUT_SIZE; i++) {
4096                 conn_walk_drain(ipst, &ipst->ips_idl_tx_list[i]);
4097         }

4099         /*
4100          * ai can be null if this is an IPv6 ill, or if the IPv4
4101          * stream is being torn down before ARP was plumbed (e.g.,
4102          * /sbin/ifconfig plumbing a stream twice, and encountering
4103          * an error
4104          */
4105         if (ai != NULL) {
4106                 ASSERT(!ill->ill_isv6);
4107                 mutex_enter(&ai->ai_lock);
4108                 ai->ai_ill = NULL;
4109                 if (ai->ai_arl == NULL) {
4110                         mutex_destroy(&ai->ai_lock);
4111                         kmem_free(ai, sizeof (*ai));
4112                 } else {
4113                         cv_signal(&ai->ai_ill_unplumb_done);
4114                         mutex_exit(&ai->ai_lock);
4115                 }
4116         }

4118         mutex_enter(&ipst->ips_ip_mi_lock);
4119         mi_close_unlink(&ipst->ips_ip_g_head, (IDP)ill);
4120         mutex_exit(&ipst->ips_ip_mi_lock);

4122         /*
4123          * credp could be null if the open didn't succeed and ip_modopen
4124          * itself calls ip_close.
4125          */
4126         if (ill->ill_credp != NULL)
4127                 crfree(ill->ill_credp);

4129         mutex_destroy(&ill->ill_saved_ire_lock);
4130         mutex_destroy(&ill->ill_lock);
4131         rw_destroy(&ill->ill_mcast_lock);
4132         mutex_destroy(&ill->ill_mcast_serializer);
4133         list_destroy(&ill->ill_nce);

4135         /*
4136          * Now we are done with the module close pieces that
4137          * need the netstack_t.
4138          */
4139         netstack_rele(ipst->ips_netstack);

4141         mi_close_free((IDP)ill);
4142         q->q_ptr = WR(q)->q_ptr = NULL;

4144         ipsq_exit(ipsq);

4146         return (0);
4147 }

4149 /*
4150  * This is called as part of close() for IP, UDP, ICMP, and RTS
4151  * in order to quiesce the conn.
4152  */
4153 void
```



```
4154 ip_quiesce_conn(conn_t *connp)
4155 {
4156         boolean_t       drain_cleanup_reqd = B_FALSE;
4157         boolean_t       conn_ioctl_cleanup_reqd = B_FALSE;
4158         boolean_t       ilg_cleanup_reqd = B_FALSE;
4159         ip_stack_t      *ipst;

4161         ASSERT(!IPCL_IS_TCP(connp));
4162         ipst = connp->conn_netstack->netstack_ip;

4164         /*
4165          * Mark the conn as closing, and this conn must not be
4166          * inserted in future into any list. Eg. conn_drain_insert(),
4167          * won't insert this conn into the conn_drain_list.
4168          *
4169          * conn_idl, and conn_ilg cannot get set henceforth.
4170          */
4171         mutex_enter(&connp->conn_lock);
4172         ASSERT(!(connp->conn_state_flags & CONN_QUIESCED));
4173         connp->conn_state_flags |= CONN_CLOSING;
4174         if (connp->conn_idl != NULL)
4175                 drain_cleanup_reqd = B_TRUE;
4176         if (connp->conn_oper_pending_ill != NULL)
4177                 conn_ioctl_cleanup_reqd = B_TRUE;
4178         if (connp->conn_dhcpinit_ill != NULL) {
4179                 ASSERT(connp->conn_dhcpinit_ill->ill_dhcpinit != 0);
4180                 atomic_dec_32(&connp->conn_dhcpinit_ill->ill_dhcpinit);
4181                 ill_set_inputfn(connp->conn_dhcpinit_ill);
4182                 connp->conn_dhcpinit_ill = NULL;
4183         }
4184         if (connp->conn_ilg != NULL)
4185                 ilg_cleanup_reqd = B_TRUE;
4186         mutex_exit(&connp->conn_lock);

4188         if (conn_ioctl_cleanup_reqd)
4189                 conn_ioctl_cleanup(connp);

4191         if (is_system_labeled() && connp->conn_anon_port) {
4192                 (void) tsol_mlp_anon(crgetzone(connp->conn_cred),
4193                     connp->conn_mlp_type, connp->conn_proto,
4194                     ntohs(connp->conn_lport), B_FALSE);
4195                 connp->conn_anon_port = 0;
4196         }
4197         connp->conn_mlp_type = mlptSingle;

4199         /*
4200          * Remove this conn from any fanout list it is on.
4201          * and then wait for any threads currently operating
4202          * on this endpoint to finish
4203          */
4204         ipcl_hash_remove(connp);

4206         /*
4207          * Remove this conn from the drain list, and do any other cleanup that
4208          * may be required.  (TCP conns are never flow controlled, and
4209          * conn_idl will be NULL.)
4210          */
4211         if (drain_cleanup_reqd && connp->conn_idl != NULL) {
4212                 idl_t *idl = connp->conn_idl;

4214                 mutex_enter(&idl->idl_lock);
4215                 conn_drain(connp, B_TRUE);
4216                 mutex_exit(&idl->idl_lock);
4217         }

4219         if (connp == ipst->ips_ip_g_mrouter)
```

```
4220                 (void) ip_mrouter_done(ipst);

4222         if (ilg_cleanup_reqd)
4223                 ilg_delete_all(connp);

4225         /*
4226          * Now conn refcnt can increase only thru CONN_INC_REF_LOCKED.
4227          * callers from write side can't be there now because close
4228          * is in progress. The only other caller is ipcl_walk
4229          * which checks for the condemned flag.
4230          */
4231         mutex_enter(&connp->conn_lock);
4232         connp->conn_state_flags |= CONN_CONDEMNED;
4233         while (connp->conn_ref != 1)
4234                 cv_wait(&connp->conn_cv, &connp->conn_lock);
4235         connp->conn_state_flags |= CONN_QUIESCED;
4236         mutex_exit(&connp->conn_lock);
4237 }

4239 /* ARGSUSED */
4240 int
4241 ip_close(queue_t *q, int flags)
4242 {
4243         conn_t          *connp;

4245         /*
4246          * Call the appropriate delete routine depending on whether this is
4247          * a module or device.
4248          */
4249         if (WR(q)->q_next != NULL) {
4250                 /* This is a module close */
4251                 return (ip_modclose((ill_t *)q->q_ptr));
4252         }

4254         connp = q->q_ptr;
4255         ip_quiesce_conn(connp);

4257         qprocsoff(q);

4259         /*
4260          * Now we are truly single threaded on this stream, and can
4261          * delete the things hanging off the connp, and finally the connp.
4262          * We removed this connp from the fanout list, it cannot be
4263          * accessed thru the fanouts, and we already waited for the
4264          * conn_ref to drop to 0. We are already in close, so
4265          * there cannot be any other thread from the top. qprocsoff
4266          * has completed, and service has completed or won't run in
4267          * future.
4268          */
4269         ASSERT(connp->conn_ref == 1);

4271         inet_minor_free(connp->conn_minor_arena, connp->conn_dev);

4273         connp->conn_ref--;
4274         ipcl_conn_destroy(connp);

4276         q->q_ptr = WR(q)->q_ptr = NULL;
4277         return (0);
4278 }

4280 /*
4281  * Wapper around putnext() so that ip_rts_request can merely use
4282  * conn_recv.
4283  */
4284 /*ARGSUSED2*/
4285 static void
```



```
4286 ip_conn_input(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
4287 {
4288         conn_t *connp = (conn_t *)arg1;

4290         putnext(connp->conn_rq, mp);
4291 }

4293 /* Dummy in case ICMP error delivery is attempted to a /dev/ip instance */
4294 /* ARGSUSED */
4295 static void
4296 ip_conn_input_icmp(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
4297 {
4298         freemsg(mp);
4299 }

4301 /*
4302  * Called when the module is about to be unloaded
4303  */
4304 void
4305 ip_ddi_destroy(void)
4306 {
4307         /* This needs to be called before destroying any transports. */
4308         mutex_enter(&cpu_lock);
4309         unregister_cpu_setup_func(ip_tp_cpu_update, NULL);
4310         mutex_exit(&cpu_lock);

4312         tnet_fini();

4314         icmp_ddi_g_destroy();
4315         rts_ddi_g_destroy();
4316         udp_ddi_g_destroy();
4317         dccp_ddi_g_destroy();
4318 #endif /* ! codereview */
4319         sctp_ddi_g_destroy();
4320         tcp_ddi_g_destroy();
4321         ilb_ddi_g_destroy();
4322         dce_g_destroy();
4323         ipsec_policy_g_destroy();
4324         ipcl_g_destroy();
4325         ip_net_g_destroy();
4326         ip_ire_g_fini();
4327         inet_minor_destroy(ip_minor_arena_sa);
4328 #if defined(_LP64)
4329         inet_minor_destroy(ip_minor_arena_la);
4330 #endif

4332 #ifdef DEBUG
4333         list_destroy(&ip_thread_list);
4334         rw_destroy(&ip_thread_rwlock);
4335         tsd_destroy(&ip_thread_data);
4336 #endif

4338         netstack_unregister(NS_IP);
4339 }

4341 /*
4342  * First step in cleanup.
4343  */
4344 /* ARGSUSED */
4345 static void
4346 ip_stack_shutdown(netstackid_t stackid, void *arg)
4347 {
4348         ip_stack_t *ipst = (ip_stack_t *)arg;

4350 #ifdef NS_DEBUG
4351         printf("ip_stack_shutdown(%p, stack %d)\n", (void *)ipst, stackid);
```

```
#endif

        /*
         * Perform cleanup for special interfaces (loopback and IPMP).
         */
        ip_interface_cleanup(ipst);

        /*
         * The *_hook_shutdown()s start the process of notifying any
         * consumers that things are going away.... nothing is destroyed.
         */
        ipv4_hook_shutdown(ipst);
        ipv6_hook_shutdown(ipst);
        arp_hook_shutdown(ipst);

        mutex_enter(&ipst->ips_capab_taskq_lock);
        ipst->ips_capab_taskq_quit = B_TRUE;
        cv_signal(&ipst->ips_capab_taskq_cv);
        mutex_exit(&ipst->ips_capab_taskq_lock);
}

/*
 * Free the IP stack instance.
 */
static void
ip_stack_fini(netstackid_t stackid, void *arg)
{
        ip_stack_t *ipst = (ip_stack_t *)arg;
        int ret;

#ifdef NS_DEBUG
        printf("ip_stack_fini(%p, stack %d)\n", (void *)ipst, stackid);
#endif
        /*
         * At this point, all of the notifications that the events and
         * protocols are going away have been run, meaning that we can
         * now set about starting to clean things up.
         */
        ipobs_fini(ipst);
        ipv4_hook_destroy(ipst);
        ipv6_hook_destroy(ipst);
        arp_hook_destroy(ipst);
        ip_net_destroy(ipst);

        ipmp_destroy(ipst);

        ip_kstat_fini(stackid, ipst->ips_ip_mibkp);
        ipst->ips_ip_mibkp = NULL;
        icmp_kstat_fini(stackid, ipst->ips_icmp_mibkp);
        ipst->ips_icmp_mibkp = NULL;
        ip_kstat2_fini(stackid, ipst->ips_ip_kstat);
        ipst->ips_ip_kstat = NULL;
        bzero(&ipst->ips_ip_statistics, sizeof (ipst->ips_ip_statistics));
        ip6_kstat_fini(stackid, ipst->ips_ip6_kstat);
        ipst->ips_ip6_kstat = NULL;
        bzero(&ipst->ips_ip6_statistics, sizeof (ipst->ips_ip6_statistics));

        kmem_free(ipst->ips_propinfo_tbl,
            ip_propinfo_count * sizeof (mod_prop_info_t));
        ipst->ips_propinfo_tbl = NULL;

        dce_stack_destroy(ipst);
        ip_mrouter_stack_destroy(ipst);

        ret = untimeout(ipst->ips_igmp_timeout_id);
        if (ret == -1) {
```



```
                ASSERT(ipst->ips_igmp_timeout_id == 0);
        } else {
                ASSERT(ipst->ips_igmp_timeout_id != 0);
                ipst->ips_igmp_timeout_id = 0;
        }
        ret = untimeout(ipst->ips_igmp_slowtimeout_id);
        if (ret == -1) {
                ASSERT(ipst->ips_igmp_slowtimeout_id == 0);
        } else {
                ASSERT(ipst->ips_igmp_slowtimeout_id != 0);
                ipst->ips_igmp_slowtimeout_id = 0;
        }
        ret = untimeout(ipst->ips_mld_timeout_id);
        if (ret == -1) {
                ASSERT(ipst->ips_mld_timeout_id == 0);
        } else {
                ASSERT(ipst->ips_mld_timeout_id != 0);
                ipst->ips_mld_timeout_id = 0;
        }
        ret = untimeout(ipst->ips_mld_slowtimeout_id);
        if (ret == -1) {
                ASSERT(ipst->ips_mld_slowtimeout_id == 0);
        } else {
                ASSERT(ipst->ips_mld_slowtimeout_id != 0);
                ipst->ips_mld_slowtimeout_id = 0;
        }

        ip_ire_fini(ipst);
        ip6_asp_free(ipst);
        conn_drain_fini(ipst);
        ipcl_destroy(ipst);

        mutex_destroy(&ipst->ips_ndp4->ndp_g_lock);
        mutex_destroy(&ipst->ips_ndp6->ndp_g_lock);
        kmem_free(ipst->ips_ndp4, sizeof (ndp_g_t));
        ipst->ips_ndp4 = NULL;
        kmem_free(ipst->ips_ndp6, sizeof (ndp_g_t));
        ipst->ips_ndp6 = NULL;

        if (ipst->ips_loopback_ksp != NULL) {
                kstat_delete_netstack(ipst->ips_loopback_ksp, stackid);
                ipst->ips_loopback_ksp = NULL;
        }

        mutex_destroy(&ipst->ips_capab_taskq_lock);
        cv_destroy(&ipst->ips_capab_taskq_cv);

        rw_destroy(&ipst->ips_srcid_lock);

        mutex_destroy(&ipst->ips_ip_mi_lock);
        rw_destroy(&ipst->ips_ill_g_usesrc_lock);

        mutex_destroy(&ipst->ips_igmp_timer_lock);
        mutex_destroy(&ipst->ips_mld_timer_lock);
        mutex_destroy(&ipst->ips_igmp_slowtimeout_lock);
        mutex_destroy(&ipst->ips_mld_slowtimeout_lock);
        mutex_destroy(&ipst->ips_ip_addr_avail_lock);
        rw_destroy(&ipst->ips_ill_g_lock);

        kmem_free(ipst->ips_phyint_g_list, sizeof (phyint_list_t));
        ipst->ips_phyint_g_list = NULL;
        kmem_free(ipst->ips_ill_g_heads, sizeof (ill_g_head_t) * MAX_G_HEADS);
        ipst->ips_ill_g_heads = NULL;

        ldi_ident_release(ipst->ips_ldi_ident);
        kmem_free(ipst, sizeof (*ipst));
```

```
4484 }

4486 /*
4487  * This function is called from the TSD destructor, and is used to debug
4488  * reference count issues in IP. See block comment in <inet/ip_if.h> for
4489  * details.
4490  */
4491 static void
4492 ip_thread_exit(void *phash)
4493 {
4494         th_hash_t *thh = phash;

4496         rw_enter(&ip_thread_rwlock, RW_WRITER);
4497         list_remove(&ip_thread_list, thh);
4498         rw_exit(&ip_thread_rwlock);
4499         mod_hash_destroy_hash(thh->thh_hash);
4500         kmem_free(thh, sizeof (*thh));
4501 }

4503 /*
4504  * Called when the IP kernel module is loaded into the kernel
4505  */
4506 void
4507 ip_ddi_init(void)
4508 {
4509         ip_squeue_flag = ip_squeue_switch(ip_squeue_enter);

4511         /*
4512          * For IP and TCP the minor numbers should start from 2 since we have 4
4513          * initial devices: ip, ip6, tcp, tcp6.
4514          */
4515         /*
4516          * If this is a 64-bit kernel, then create two separate arenas -
4517          * one for TLIs in the range of INET_MIN_DEV+2 through 2^^18-1, and the
4518          * other for socket apps in the range 2^^18 through 2^^32-1.
4519          */
4520         ip_minor_arena_la = NULL;
4521         ip_minor_arena_sa = NULL;
4522 #if defined(_LP64)
4523         if ((ip_minor_arena_sa = inet_minor_create("ip_minor_arena_sa",
4524             INET_MIN_DEV + 2, MAXMIN32, KM_SLEEP)) == NULL) {
4525                 cmn_err(CE_PANIC,
4526                     "ip_ddi_init: ip_minor_arena_sa creation failed\n");
4527         }
4528         if ((ip_minor_arena_la = inet_minor_create("ip_minor_arena_la",
4529             MAXMIN32 + 1, MAXMIN64, KM_SLEEP)) == NULL) {
4530                 cmn_err(CE_PANIC,
4531                     "ip_ddi_init: ip_minor_arena_la creation failed\n");
4532         }
4533 #else
4534         if ((ip_minor_arena_sa = inet_minor_create("ip_minor_arena_sa",
4535             INET_MIN_DEV + 2, MAXMIN, KM_SLEEP)) == NULL) {
4536                 cmn_err(CE_PANIC,
4537                     "ip_ddi_init: ip_minor_arena_sa creation failed\n");
4538         }
4539 #endif
4540         ip_poll_normal_ticks = MSEC_TO_TICK_ROUNDUP(ip_poll_normal_ms);

4542         ipcl_g_init();
4543         ip_ire_g_init();
4544         ip_net_g_init();

4546 #ifdef DEBUG
4547         tsd_create(&ip_thread_data, ip_thread_exit);
4548         rw_init(&ip_thread_rwlock, NULL, RW_DEFAULT, NULL);
4549         list_create(&ip_thread_list, sizeof (th_hash_t),
```



```
4550             offsetof(th_hash_t, thh_link));
4551 #endif
4552         ipsec_policy_g_init();
4553         tcp_ddi_g_init();
4554         sctp_ddi_g_init();
4555         dccp_ddi_g_init();
4556 #endif /* ! codereview */
4557         dce_g_init();

4559         /*
4560          * We want to be informed each time a stack is created or
4561          * destroyed in the kernel, so we can maintain the
4562          * set of udp_stack_t's.
4563          */
4564         netstack_register(NS_IP, ip_stack_init, ip_stack_shutdown,
4565             ip_stack_fini);

4567         tnet_init();

4569         udp_ddi_g_init();
4570         rts_ddi_g_init();
4571         icmp_ddi_g_init();
4572         ilb_ddi_g_init();

4574         /* This needs to be called after all transports are initialized. */
4575         mutex_enter(&cpu_lock);
4576         register_cpu_setup_func(ip_tp_cpu_update, NULL);
4577         mutex_exit(&cpu_lock);
4578 }

4580 /*
4581  * Initialize the IP stack instance.
4582  */
4583 static void *
4584 ip_stack_init(netstackid_t stackid, netstack_t *ns)
4585 {
4586         ip_stack_t      *ipst;
4587         size_t          arrsz;
4588         major_t         major;

4590 #ifdef NS_DEBUG
4591         printf("ip_stack_init(stack %d)\n", stackid);
4592 #endif

4594         ipst = (ip_stack_t *)kmem_zalloc(sizeof (*ipst), KM_SLEEP);
4595         ipst->ips_netstack = ns;

4597         ipst->ips_ill_g_heads = kmem_zalloc(sizeof (ill_g_head_t) * MAX_G_HEADS,
4598             KM_SLEEP);
4599         ipst->ips_phyint_g_list = kmem_zalloc(sizeof (phyint_list_t),
4600             KM_SLEEP);
4601         ipst->ips_ndp4 = kmem_zalloc(sizeof (ndp_g_t), KM_SLEEP);
4602         ipst->ips_ndp6 = kmem_zalloc(sizeof (ndp_g_t), KM_SLEEP);
4603         mutex_init(&ipst->ips_ndp4->ndp_g_lock, NULL, MUTEX_DEFAULT, NULL);
4604         mutex_init(&ipst->ips_ndp6->ndp_g_lock, NULL, MUTEX_DEFAULT, NULL);

4606         mutex_init(&ipst->ips_igmp_timer_lock, NULL, MUTEX_DEFAULT, NULL);
4607         ipst->ips_igmp_deferred_next = INFINITY;
4608         mutex_init(&ipst->ips_mld_timer_lock, NULL, MUTEX_DEFAULT, NULL);
4609         ipst->ips_mld_deferred_next = INFINITY;
4610         mutex_init(&ipst->ips_igmp_slowtimeout_lock, NULL, MUTEX_DEFAULT, NULL);
4611         mutex_init(&ipst->ips_mld_slowtimeout_lock, NULL, MUTEX_DEFAULT, NULL);
4612         mutex_init(&ipst->ips_ip_mi_lock, NULL, MUTEX_DEFAULT, NULL);
4613         mutex_init(&ipst->ips_ip_addr_avail_lock, NULL, MUTEX_DEFAULT, NULL);
4614         rw_init(&ipst->ips_ill_g_lock, NULL, RW_DEFAULT, NULL);
4615         rw_init(&ipst->ips_ill_g_usesrc_lock, NULL, RW_DEFAULT, NULL);
```

```
4617         ipcl_init(ipst);
4618         ip_ire_init(ipst);
4619         ip6_asp_init(ipst);
4620         ipif_init(ipst);
4621         conn_drain_init(ipst);
4622         ip_mrouter_stack_init(ipst);
4623         dce_stack_init(ipst);

4625         ipst->ips_ip_multirt_log_interval = 1000;

4627         ipst->ips_ill_index = 1;

4629         ipst->ips_saved_ip_forwarding = -1;
4630         ipst->ips_reg_vif_num = ALL_VIFS;       /* Index to Register vif */

4632         arrsz = ip_propinfo_count * sizeof (mod_prop_info_t);
4633         ipst->ips_propinfo_tbl = (mod_prop_info_t *)kmem_alloc(arrsz, KM_SLEEP);
4634         bcopy(ip_propinfo_tbl, ipst->ips_propinfo_tbl, arrsz);

4636         ipst->ips_ip_mibkp = ip_kstat_init(stackid, ipst);
4637         ipst->ips_icmp_mibkp = icmp_kstat_init(stackid);
4638         ipst->ips_ip_kstat = ip_kstat2_init(stackid, &ipst->ips_ip_statistics);
4639         ipst->ips_ip6_kstat =
4640             ip6_kstat_init(stackid, &ipst->ips_ip6_statistics);

4642         ipst->ips_ip_src_id = 1;
4643         rw_init(&ipst->ips_srcid_lock, NULL, RW_DEFAULT, NULL);

4645         ipst->ips_src_generation = SRC_GENERATION_INITIAL;

4647         ip_net_init(ipst, ns);
4648         ipv4_hook_init(ipst);
4649         ipv6_hook_init(ipst);
4650         arp_hook_init(ipst);
4651         ipmp_init(ipst);
4652         ipobs_init(ipst);

4654         /*
4655          * Create the taskq dispatcher thread and initialize related stuff.
4656          */
4657         mutex_init(&ipst->ips_capab_taskq_lock, NULL, MUTEX_DEFAULT, NULL);
4658         cv_init(&ipst->ips_capab_taskq_cv, NULL, CV_DEFAULT, NULL);
4659         ipst->ips_capab_taskq_thread = thread_create(NULL, 0,
4660             ill_taskq_dispatch, ipst, 0, &p0, TS_RUN, minclsyspri);

4662         major = mod_name_to_major(INET_NAME);
4663         (void) ldi_ident_from_major(major, &ipst->ips_ldi_ident);
4664         return (ipst);
4665 }

4667 /*
4668  * Allocate and initialize a DLPI template of the specified length.  (May be
4669  * called as writer.)
4670  */
4671 mblk_t *
4672 ip_dlpi_alloc(size_t len, t_uscalar_t prim)
4673 {
4674         mblk_t  *mp;

4676         mp = allocb(len, BPRI_MED);
4677         if (!mp)
4678                 return (NULL);

4680         /*
4681          * DLPIv2 says that DL_INFO_REQ and DL_TOKEN_REQ (the latter
```



```
4682          * of which we don't seem to use) are sent with M_PCPROTO, and
4683          * that other DLPI are M_PROTO.
4684          */
4685         if (prim == DL_INFO_REQ) {
4686                 mp->b_datap->db_type = M_PCPROTO;
4687         } else {
4688                 mp->b_datap->db_type = M_PROTO;
4689         }

4691         mp->b_wptr = mp->b_rptr + len;
4692         bzero(mp->b_rptr, len);
4693         ((dl_unitdata_req_t *)mp->b_rptr)->dl_primitive = prim;
4694         return (mp);
4695 }

4697 /*
4698  * Allocate and initialize a DLPI notification.  (May be called as writer.)
4699  */
4700 mblk_t *
4701 ip_dlnotify_alloc(uint_t notification, uint_t data)
4702 {
4703         dl_notify_ind_t *notifyp;
4704         mblk_t          *mp;

4706         if ((mp = ip_dlpi_alloc(DL_NOTIFY_IND_SIZE, DL_NOTIFY_IND)) == NULL)
4707                 return (NULL);

4709         notifyp = (dl_notify_ind_t *)mp->b_rptr;
4710         notifyp->dl_notification = notification;
4711         notifyp->dl_data = data;
4712         return (mp);
4713 }

4715 mblk_t *
4716 ip_dlnotify_alloc2(uint_t notification, uint_t data1, uint_t data2)
4717 {
4718         dl_notify_ind_t *notifyp;
4719         mblk_t          *mp;

4721         if ((mp = ip_dlpi_alloc(DL_NOTIFY_IND_SIZE, DL_NOTIFY_IND)) == NULL)
4722                 return (NULL);

4724         notifyp = (dl_notify_ind_t *)mp->b_rptr;
4725         notifyp->dl_notification = notification;
4726         notifyp->dl_data1 = data1;
4727         notifyp->dl_data2 = data2;
4728         return (mp);
4729 }

4731 /*
4732  * Debug formatting routine.  Returns a character string representation of the
4733  * addr in buf, of the form xxx.xxx.xxx.xxx.  This routine takes the address
4734  * in the form of a ipaddr_t and calls ip_dot_saddr with a pointer.
4735  *
4736  * Once the ndd table-printing interfaces are removed, this can be changed to
4737  * standard dotted-decimal form.
4738  */
4739 char *
4740 ip_dot_addr(ipaddr_t addr, char *buf)
4741 {
4742         uint8_t *ap = (uint8_t *)&addr;

4744         (void) mi_sprintf(buf, "%03d.%03d.%03d.%03d",
4745             ap[0] & 0xFF, ap[1] & 0xFF, ap[2] & 0xFF, ap[3] & 0xFF);
4746         return (buf);
4747 }
```

```
4749 /*
4750  * Write the given MAC address as a printable string in the usual colon-
4751  * separated format.
4752  */
4753 const char *
4754 mac_colon_addr(const uint8_t *addr, size_t alen, char *buf, size_t buflen)
4755 {
4756         char *bp;

4758         if (alen == 0 || buflen < 4)
4759                 return ("?");
4760         bp = buf;
4761         for (;;) {
4762                 /*
4763                  * If there are more MAC address bytes available, but we won't
4764                  * have any room to print them, then add "..." to the string
4765                  * instead.  See below for the 'magic number' explanation.
4766                  */
4767                 if ((alen == 2 && buflen < 6) || (alen > 2 && buflen < 7)) {
4768                         (void) strcpy(bp, "...");
4769                         break;
4770                 }
4771                 (void) sprintf(bp, "%02x", *addr++);
4772                 bp += 2;
4773                 if (--alen == 0)
4774                         break;
4775                 *bp++ = ':';
4776                 buflen -= 3;
4777                 /*
4778                  * At this point, based on the first 'if' statement above,
4779                  * either alen == 1 and buflen >= 3, or alen > 1 and
4780                  * buflen >= 4.  The first case leaves room for the final "xx"
4781                  * number and trailing NUL byte.  The second leaves room for at
4782                  * least "...".  Thus the apparently 'magic' numbers chosen for
4783                  * that statement.
4784                  */
4785         }
4786         return (buf);
4787 }

4789 /*
4790  * Called when it is conceptually a ULP that would sent the packet
4791  * e.g., port unreachable and protocol unreachable. Check that the packet
4792  * would have passed the IPsec global policy before sending the error.
4793  *
4794  * Send an ICMP error after patching up the packet appropriately.
4795  * Uses ip_drop_input and bumps the appropriate MIB.
4796  */
4797 void
4798 ip_fanout_send_icmp_v4(mblk_t *mp, uint_t icmp_type, uint_t icmp_code,
4799     ip_recv_attr_t *ira)
4800 {
4801         ipha_t          *ipha;
4802         boolean_t       secure;
4803         ill_t           *ill = ira->ira_ill;
4804         ip_stack_t      *ipst = ill->ill_ipst;
4805         netstack_t      *ns = ipst->ips_netstack;
4806         ipsec_stack_t   *ipss = ns->netstack_ipsec;

4808         secure = ira->ira_flags & IRAF_IPSEC_SECURE;

4810         /*
4811          * We are generating an icmp error for some inbound packet.
4812          * Called from all ip_fanout_(udp, tcp, proto) functions.
4813          * Before we generate an error, check with global policy
```



```
4814          * to see whether this is allowed to enter the system. As
4815          * there is no "conn", we are checking with global policy.
4816          */
4817         ipha = (ipha_t *)mp->b_rptr;
4818         if (secure || ipss->ipsec_inbound_v4_policy_present) {
4819                 mp = ipsec_check_global_policy(mp, NULL, ipha, NULL, ira, ns);
4820                 if (mp == NULL)
4821                         return;
4822         }

4824         /* We never send errors for protocols that we do implement */
4825         if (ira->ira_protocol == IPPROTO_ICMP ||
4826             ira->ira_protocol == IPPROTO_IGMP) {
4827                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
4828                 ip_drop_input("ip_fanout_send_icmp_v4", mp, ill);
4829                 freemsg(mp);
4830                 return;
4831         }
4832         /*
4833          * Have to correct checksum since
4834          * the packet might have been
4835          * fragmented and the reassembly code in ip_rput
4836          * does not restore the IP checksum.
4837          */
4838         ipha->ipha_hdr_checksum = 0;
4839         ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

4841         switch (icmp_type) {
4842         case ICMP_DEST_UNREACHABLE:
4843                 switch (icmp_code) {
4844                 case ICMP_PROTOCOL_UNREACHABLE:
4845                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInUnknownProtos);
4846                         ip_drop_input("ipIfStatsInUnknownProtos", mp, ill);
4847                         break;
4848                 case ICMP_PORT_UNREACHABLE:
4849                         BUMP_MIB(ill->ill_ip_mib, udpIfStatsNoPorts);
4850                         ip_drop_input("ipIfStatsNoPorts", mp, ill);
4851                         break;
4852                 }

4854                 icmp_unreachable(mp, icmp_code, ira);
4855                 break;
4856         default:
4857 #ifdef DEBUG
4858                 panic("ip_fanout_send_icmp_v4: wrong type");
4859                 /*NOTREACHED*/
4860 #else
4861                 freemsg(mp);
4862                 break;
4863 #endif
4864         }
4865 }

4867 /*
4868  * Used to send an ICMP error message when a packet is received for
4869  * a protocol that is not supported. The mblk passed as argument
4870  * is consumed by this function.
4871  */
4872 void
4873 ip_proto_not_sup(mblk_t *mp, ip_recv_attr_t *ira)
4874 {
4875         ipha_t          *ipha;

4877         ipha = (ipha_t *)mp->b_rptr;
4878         if (ira->ira_flags & IRAF_IS_IPV4) {
4879                 ASSERT(IPH_HDR_VERSION(ipha) == IP_VERSION);
```

```
4880                 ip_fanout_send_icmp_v4(mp, ICMP_DEST_UNREACHABLE,
4881                     ICMP_PROTOCOL_UNREACHABLE, ira);
4882         } else {
4883                 ASSERT(IPH_HDR_VERSION(ipha) == IPV6_VERSION);
4884                 ip_fanout_send_icmp_v6(mp, ICMP6_PARAM_PROB,
4885                     ICMP6_PARAMPROB_NEXTHEADER, ira);
4886         }
4887 }

4889 /*
4890  * Deliver a rawip packet to the given conn, possibly applying ipsec policy.
4891  * Handles IPv4 and IPv6.
4892  * We are responsible for disposing of mp, such as by freemsg() or putnext()
4893  * Caller is responsible for dropping references to the conn.
4894  */
4895 void
4896 ip_fanout_proto_conn(conn_t *connp, mblk_t *mp, ipha_t *ipha, ip6_t *ip6h,
4897     ip_recv_attr_t *ira)
4898 {
4899         ill_t           *ill = ira->ira_ill;
4900         ip_stack_t      *ipst = ill->ill_ipst;
4901         ipsec_stack_t   *ipss = ipst->ips_netstack->netstack_ipsec;
4902         boolean_t       secure;
4903         uint_t          protocol = ira->ira_protocol;
4904         iaflags_t       iraflags = ira->ira_flags;
4905         queue_t         *rq;

4907         secure = iraflags & IRAF_IPSEC_SECURE;

4909         rq = connp->conn_rq;
4910         if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrld : !canputnext(rq)) {
4911                 switch (protocol) {
4912                 case IPPROTO_ICMPV6:
4913                         BUMP_MIB(ill->ill_icmp6_mib, ipv6IfIcmpInOverflows);
4914                         break;
4915                 case IPPROTO_ICMP:
4916                         BUMP_MIB(&ipst->ips_icmp_mib, icmpInOverflows);
4917                         break;
4918                 default:
4919                         BUMP_MIB(ill->ill_ip_mib, rawipIfStatsInOverflows);
4920                         break;
4921                 }
4922                 freemsg(mp);
4923                 return;
4924         }

4926         ASSERT(!(IPCL_IS_IPTUN(connp)));

4928         if (((iraflags & IRAF_IS_IPV4) ?
4929             CONN_INBOUND_POLICY_PRESENT(connp, ipss) :
4930             CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)) ||
4931             secure) {
4932                 mp = ipsec_check_inbound_policy(mp, connp, ipha,
4933                     ip6h, ira);
4934                 if (mp == NULL) {
4935                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
4936                         /* Note that mp is NULL */
4937                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
4938                         return;
4939                 }
4940         }

4942         if (iraflags & IRAF_ICMP_ERROR) {
4943                 (connp->conn_recvicmp)(connp, mp, NULL, ira);
4944         } else {
4945                 ill_t *rill = ira->ira_rill;
```



```
4947                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
4948                 ira->ira_ill = ira->ira_rill = NULL;
4949                 /* Send it upstream */
4950                 (connp->conn_recv)(connp, mp, NULL, ira);
4951                 ira->ira_ill = ill;
4952                 ira->ira_rill = rill;
4953         }
4954 }

4956 /*
4957  * Handle protocols with which IP is less intimate.  There
4958  * can be more than one stream bound to a particular
4959  * protocol.  When this is the case, normally each one gets a copy
4960  * of any incoming packets.
4961  *
4962  * IPsec NOTE :
4963  *
4964  * Don't allow a secure packet going up a non-secure connection.
4965  * We don't allow this because
4966  *
4967  * 1) Reply might go out in clear which will be dropped at
4968  *    the sending side.
4969  * 2) If the reply goes out in clear it will give the
4970  *    adversary enough information for getting the key in
4971  *    most of the cases.
4972  *
4973  * Moreover getting a secure packet when we expect clear
4974  * implies that SA's were added without checking for
4975  * policy on both ends. This should not happen once ISAKMP
4976  * is used to negotiate SAs as SAs will be added only after
4977  * verifying the policy.
4978  *
4979  * Zones notes:
4980  * Earlier in ip_input on a system with multiple shared-IP zones we
4981  * duplicate the multicast and broadcast packets and send them up
4982  * with each explicit zoneid that exists on that ill.
4983  * This means that here we can match the zoneid with SO_ALLZONES being special.
4984  */
4985 void
4986 ip_fanout_proto_v4(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
4987 {
4988         mblk_t          *mp1;
4989         ipaddr_t        laddr;
4990         conn_t          *connp, *first_connp, *next_connp;
4991         connf_t         *connfp;
4992         ill_t           *ill = ira->ira_ill;
4993         ip_stack_t      *ipst = ill->ill_ipst;

4995         laddr = ipha->ipha_dst;

4997         connfp = &ipst->ips_ipcl_proto_fanout_v4[ira->ira_protocol];
4998         mutex_enter(&connfp->connf_lock);
4999         connp = connfp->connf_head;
5000         for (connp = connfp->connf_head; connp != NULL;
5001             connp = connp->conn_next) {
5002                 /* Note: IPCL_PROTO_MATCH includes conn_wantpacket */
5003                 if (IPCL_PROTO_MATCH(connp, ira, ipha) &&
5004                     (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5005                     tsol_receive_local(mp, &laddr, IPV4_VERSION, ira, connp))) {
5006                         break;
5007                 }
5008         }

5010         if (connp == NULL) {
5011                 /*
```

```
5012                 * No one bound to these addresses.  Is
5013                 * there a client that wants all
5014                 * unclaimed datagrams?
5015                 */
5016                mutex_exit(&connfp->connf_lock);
5017                ip_fanout_send_icmp_v4(mp, ICMP_DEST_UNREACHABLE,
5018                    ICMP_PROTOCOL_UNREACHABLE, ira);
5019                return;
5020        }

5022        ASSERT(IPCL_IS_NONSTR(connp) || connp->conn_rq != NULL);

5024        CONN_INC_REF(connp);
5025        first_connp = connp;
5026        connp = connp->conn_next;

5028        for (;;) {
5029                while (connp != NULL) {
5030                        /* Note: IPCL_PROTO_MATCH includes conn_wantpacket */
5031                        if (IPCL_PROTO_MATCH(connp, ira, ipha) &&
5032                            (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5033                            tsol_receive_local(mp, &laddr, IPV4_VERSION,
5034                            ira, connp)))
5035                                break;
5036                        connp = connp->conn_next;
5037                }

5039                if (connp == NULL) {
5040                        /* No more interested clients */
5041                        connp = first_connp;
5042                        break;
5043                }
5044                if (((mp1 = dupmsg(mp)) == NULL) &&
5045                    ((mp1 = copymsg(mp)) == NULL)) {
5046                        /* Memory allocation failed */
5047                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5048                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
5049                        connp = first_connp;
5050                        break;
5051                }

5053                CONN_INC_REF(connp);
5054                mutex_exit(&connfp->connf_lock);

5056                ip_fanout_proto_conn(connp, mp1, (ipha_t *)mp1->b_rptr, NULL,
5057                    ira);

5059                mutex_enter(&connfp->connf_lock);
5060                /* Follow the next pointer before releasing the conn. */
5061                next_connp = connp->conn_next;
5062                CONN_DEC_REF(connp);
5063                connp = next_connp;
5064        }

5066        /* Last one.  Send it upstream. */
5067        mutex_exit(&connfp->connf_lock);

5069        ip_fanout_proto_conn(connp, mp, ipha, NULL, ira);

5071        CONN_DEC_REF(connp);
5072 }

5074 /*
5075  * If we have a IPsec NAT-Traversal packet, strip the zero-SPI or
5076  * pass it along to ESP if the SPI is non-zero.  Returns the mblk if the mblk
5077  * is not consumed.
```



```
5078  *
5079  * One of three things can happen, all of which affect the passed-in mblk:
5080  *
5081  * 1.) The packet is stock UDP and gets its zero-SPI stripped.  Return mblk..
5082  *
5083  * 2.) The packet is ESP-in-UDP, gets transformed into an equivalent
5084  *     ESP packet, and is passed along to ESP for consumption.  Return NULL.
5085  *
5086  * 3.) The packet is an ESP-in-UDP Keepalive.  Drop it and return NULL.
5087  */
5088 mblk_t *
5089 zero_spi_check(mblk_t *mp, ip_recv_attr_t *ira)
5090 {
5091        int shift, plen, iph_len;
5092        ipha_t *ipha;
5093        udpha_t *udpha;
5094        uint32_t *spi;
5095        uint32_t esp_ports;
5096        uint8_t *orptr;
5097        ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
5098        ipsec_stack_t   *ipss = ipst->ips_netstack->netstack_ipsec;

5100        ipha = (ipha_t *)mp->b_rptr;
5101        iph_len = ira->ira_ip_hdr_length;
5102        plen = ira->ira_pktlen;

5104        if (plen - iph_len - sizeof (udpha_t) < sizeof (uint32_t)) {
5105                /*
5106                 * Most likely a keepalive for the benefit of an intervening
5107                 * NAT.  These aren't for us, per se, so drop it.
5108                 *
5109                 * RFC 3947/8 doesn't say for sure what to do for 2-3
5110                 * byte packets (keepalives are 1-byte), but we'll drop them
5111                 * also.
5112                 */
5113                ip_drop_packet(mp, B_TRUE, ira->ira_ill,
5114                    DROPPER(ipss, ipds_esp_nat_t_ka), &ipss->ipsec_dropper);
5115                return (NULL);
5116        }

5118        if (MBLKL(mp) < iph_len + sizeof (udpha_t) + sizeof (*spi)) {
5119                /* might as well pull it all up - it might be ESP. */
5120                if (!pullupmsg(mp, -1)) {
5121                        ip_drop_packet(mp, B_TRUE, ira->ira_ill,
5122                            DROPPER(ipss, ipds_esp_nomem),
5123                            &ipss->ipsec_dropper);
5124                        return (NULL);
5125                }

5127                ipha = (ipha_t *)mp->b_rptr;
5128        }
5129        spi = (uint32_t *)(mp->b_rptr + iph_len + sizeof (udpha_t));
5130        if (*spi == 0) {
5131                /* UDP packet - remove 0-spi. */
5132                shift = sizeof (uint32_t);
5133        } else {
5134                /* ESP-in-UDP packet - reduce to ESP. */
5135                ipha->ipha_protocol = IPPROTO_ESP;
5136                shift = sizeof (udpha_t);
5137        }

5139        /* Fix IP header */
5140        ira->ira_pktlen = (plen - shift);
5141        ipha->ipha_length = htons(ira->ira_pktlen);
5142        ipha->ipha_hdr_checksum = 0;
```

```
5144         orptr = mp->b_rptr;
5145         mp->b_rptr += shift;

5147         udpha = (udpha_t *)(orptr + iph_len);
5148         if (*spi == 0) {
5149                 ASSERT((uint8_t *)ipha == orptr);
5150                 udpha->uha_length = htons(plen - shift - iph_len);
5151                 iph_len += sizeof (udpha_t);    /* For the call to ovbcopy(). */
5152                 esp_ports = 0;
5153         } else {
5154                 esp_ports = *((uint32_t *)udpha);
5155                 ASSERT(esp_ports != 0);
5156         }
5157         ovbcopy(orptr, orptr + shift, iph_len);
5158         if (esp_ports != 0) /* Punt up for ESP processing. */ {
5159                 ipha = (ipha_t *)(orptr + shift);

5161                 ira->ira_flags |= IRAF_ESP_UDP_PORTS;
5162                 ira->ira_esp_udp_ports = esp_ports;
5163                 ip_fanout_v4(mp, ipha, ira);
5164                 return (NULL);
5165         }
5166         return (mp);
5167 }

5169 /*
5170  * Deliver a udp packet to the given conn, possibly applying ipsec policy.
5171  * Handles IPv4 and IPv6.
5172  * We are responsible for disposing of mp, such as by freemsg() or putnext()
5173  * Caller is responsible for dropping references to the conn.
5174  */
5175 void
5176 ip_fanout_udp_conn(conn_t *connp, mblk_t *mp, ipha_t *ipha, ip6_t *ip6h,
5177     ip_recv_attr_t *ira)
5178 {
5179         ill_t           *ill = ira->ira_ill;
5180         ip_stack_t      *ipst = ill->ill_ipst;
5181         ipsec_stack_t   *ipss = ipst->ips_netstack->netstack_ipsec;
5182         boolean_t       secure;
5183         iaflags_t       iraflags = ira->ira_flags;

5185         secure = iraflags & IRAF_IPSEC_SECURE;

5187         if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrld :
5188             !canputnext(connp->conn_rq)) {
5189                 BUMP_MIB(ill->ill_ip_mib, udpIfStatsInOverflows);
5190                 freemsg(mp);
5191                 return;
5192         }

5194         if (((iraflags & IRAF_IS_IPV4) ?
5195             CONN_INBOUND_POLICY_PRESENT(connp, ipss) :
5196             CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)) ||
5197             secure) {
5198                 mp = ipsec_check_inbound_policy(mp, connp, ipha,
5199                     ip6h, ira);
5200                 if (mp == NULL) {
5201                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5202                         /* Note that mp is NULL */
5203                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
5204                         return;
5205                 }
5206         }

5208         /*
5209          * Since this code is not used for UDP unicast we don't need a NAT_T
```



```
5210          * check. Only ip_fanout_v4 has that check.
5211          */
5212         if (ira->ira_flags & IRAF_ICMP_ERROR) {
5213                 (connp->conn_recvicmp)(connp, mp, NULL, ira);
5214         } else {
5215                 ill_t *rill = ira->ira_rill;

5217                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
5218                 ira->ira_ill = ira->ira_rill = NULL;
5219                 /* Send it upstream */
5220                 (connp->conn_recv)(connp, mp, NULL, ira);
5221                 ira->ira_ill = ill;
5222                 ira->ira_rill = rill;
5223         }
5224 }

5226 /*
5227  * Fanout for UDP packets that are multicast or broadcast, and ICMP errors.
5228  * (Unicast fanout is handled in ip_input_v4.)
5229  *
5230  * If SO_REUSEADDR is set all multicast and broadcast packets
5231  * will be delivered to all conns bound to the same port.
5232  *
5233  * If there is at least one matching AF_INET receiver, then we will
5234  * ignore any AF_INET6 receivers.
5235  * In the special case where an AF_INET socket binds to 0.0.0.0/<port> and an
5236  * AF_INET6 socket binds to ::/<port>, only the AF_INET socket receives the IPv4
5237  * packets.
5238  *
5239  * Zones notes:
5240  * Earlier in ip_input on a system with multiple shared-IP zones we
5241  * duplicate the multicast and broadcast packets and send them up
5242  * with each explicit zoneid that exists on that ill.
5243  * This means that here we can match the zoneid with SO_ALLZONES being special.
5244  */
5245 void
5246 ip_fanout_udp_multi_v4(mblk_t *mp, ipha_t *ipha, uint16_t lport, uint16_t fport,
5247     ip_recv_attr_t *ira)
5248 {
5249         ipaddr_t        laddr;
5250         in6_addr_t      v6faddr;
5251         conn_t          *connp;
5252         connf_t         *connfp;
5253         ipaddr_t        faddr;
5254         ill_t           *ill = ira->ira_ill;
5255         ip_stack_t      *ipst = ill->ill_ipst;

5257         ASSERT(ira->ira_flags & (IRAF_MULTIBROADCAST|IRAF_ICMP_ERROR));

5259         laddr = ipha->ipha_dst;
5260         faddr = ipha->ipha_src;

5262         connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
5263         mutex_enter(&connfp->connf_lock);
5264         connp = connfp->connf_head;

5266         /*
5267          * If SO_REUSEADDR has been set on the first we send the
5268          * packet to all clients that have joined the group and
5269          * match the port.
5270          */
5271         while (connp != NULL) {
5272                 if ((IPCL_UDP_MATCH(connp, lport, laddr, fport, faddr)) &&
5273                     conn_wantpacket(connp, ira, ipha) &&
5274                     (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5275                     tsol_receive_local(mp, &laddr, IPV4_VERSION, ira, connp)))
```

```
5276                        break;
5277                connp = connp->conn_next;
5278        }

5280        if (connp == NULL)
5281                goto notfound;

5283        CONN_INC_REF(connp);

5285        if (connp->conn_reuseaddr) {
5286                conn_t          *first_connp = connp;
5287                conn_t          *next_connp;
5288                mblk_t          *mp1;

5290                connp = connp->conn_next;
5291                for (;;) {
5292                        while (connp != NULL) {
5293                                if (IPCL_UDP_MATCH(connp, lport, laddr,
5294                                    fport, faddr) &&
5295                                    conn_wantpacket(connp, ira, ipha) &&
5296                                    (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5297                                    tsol_receive_local(mp, &laddr, IPV4_VERSION,
5298                                    ira, connp)))
5299                                        break;
5300                                connp = connp->conn_next;
5301                        }
5302                        if (connp == NULL) {
5303                                /* No more interested clients */
5304                                connp = first_connp;
5305                                break;
5306                        }
5307                        if (((mp1 = dupmsg(mp)) == NULL) &&
5308                            ((mp1 = copymsg(mp)) == NULL)) {
5309                                /* Memory allocation failed */
5310                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5311                                ip_drop_input("ipIfStatsInDiscards", mp, ill);
5312                                connp = first_connp;
5313                                break;
5314                        }
5315                        CONN_INC_REF(connp);
5316                        mutex_exit(&connfp->connf_lock);

5318                        IP_STAT(ipst, ip_udp_fanmb);
5319                        ip_fanout_udp_conn(connp, mp1, (ipha_t *)mp1->b_rptr,
5320                            NULL, ira);
5321                        mutex_enter(&connfp->connf_lock);
5322                        /* Follow the next pointer before releasing the conn */
5323                        next_connp = connp->conn_next;
5324                        CONN_DEC_REF(connp);
5325                        connp = next_connp;
5326                }
5327        }

5329        /* Last one.  Send it upstream. */
5330        mutex_exit(&connfp->connf_lock);
5331        IP_STAT(ipst, ip_udp_fanmb);
5332        ip_fanout_udp_conn(connp, mp, ipha, NULL, ira);
5333        CONN_DEC_REF(connp);
5334        return;

5336 notfound:
5337        mutex_exit(&connfp->connf_lock);
5338        /*
5339         * IPv6 endpoints bound to multicast IPv4-mapped addresses
5340         * have already been matched above, since they live in the IPv4
5341         * fanout tables. This implies we only need to
```



```
5342         * check for IPv6 in6addr_any endpoints here.
5343         * Thus we compare using ipv6_all_zeros instead of the destination
5344         * address, except for the multicast group membership lookup which
5345         * uses the IPv4 destination.
5346         */
5347        IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &v6faddr);
5348        connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
5349        mutex_enter(&connfp->connf_lock);
5350        connp = connfp->connf_head;
5351        /*
5352         * IPv4 multicast packet being delivered to an AF_INET6
5353         * in6addr_any endpoint.
5354         * Need to check conn_wantpacket(). Note that we use conn_wantpacket()
5355         * and not conn_wantpacket_v6() since any multicast membership is
5356         * for an IPv4-mapped multicast address.
5357         */
5358        while (connp != NULL) {
5359                if (IPCL_UDP_MATCH_V6(connp, lport, ipv6_all_zeros,
5360                    fport, v6faddr) &&
5361                    conn_wantpacket(connp, ira, ipha) &&
5362                    (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5363                    tsol_receive_local(mp, &laddr, IPV4_VERSION, ira, connp)))
5364                        break;
5365                connp = connp->conn_next;
5366        }

5368        if (connp == NULL) {
5369                /*
5370                 * No one bound to this port.  Is
5371                 * there a client that wants all
5372                 * unclaimed datagrams?
5373                 */
5374                mutex_exit(&connfp->connf_lock);

5376                if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_UDP].connf_head !=
5377                    NULL) {
5378                        ASSERT(ira->ira_protocol == IPPROTO_UDP);
5379                        ip_fanout_proto_v4(mp, ipha, ira);
5380                } else {
5381                        /*
5382                         * We used to attempt to send an icmp error here, but
5383                         * since this is known to be a multicast packet
5384                         * and we don't send icmp errors in response to
5385                         * multicast, just drop the packet and give up sooner.
5386                         */
5387                        BUMP_MIB(ill->ill_ip_mib, udpIfStatsNoPorts);
5388                        freemsg(mp);
5389                }
5390                return;
5391        }
5392        ASSERT(IPCL_IS_NONSTR(connp) || connp->conn_rq != NULL);

5394        /*
5395         * If SO_REUSEADDR has been set on the first we send the
5396         * packet to all clients that have joined the group and
5397         * match the port.
5398         */
5399        if (connp->conn_reuseaddr) {
5400                conn_t          *first_connp = connp;
5401                conn_t          *next_connp;
5402                mblk_t          *mp1;

5404                CONN_INC_REF(connp);
5405                connp = connp->conn_next;
5406                for (;;) {
5407                        while (connp != NULL) {
```

```
5408                                if (IPCL_UDP_MATCH_V6(connp, lport,
5409                                    ipv6_all_zeros, fport, v6faddr) &&
5410                                    conn_wantpacket(connp, ira, ipha) &&
5411                                    (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5412                                    tsol_receive_local(mp, &laddr, IPV4_VERSION,
5413                                    ira, connp)))
5414                                        break;
5415                                connp = connp->conn_next;
5416                        }
5417                        if (connp == NULL) {
5418                                /* No more interested clients */
5419                                connp = first_connp;
5420                                break;
5421                        }
5422                        if (((mp1 = dupmsg(mp)) == NULL) &&
5423                            ((mp1 = copymsg(mp)) == NULL)) {
5424                                /* Memory allocation failed */
5425                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5426                                ip_drop_input("ipIfStatsInDiscards", mp, ill);
5427                                connp = first_connp;
5428                                break;
5429                        }
5430                        CONN_INC_REF(connp);
5431                        mutex_exit(&connfp->connf_lock);

5433                        IP_STAT(ipst, ip_udp_fanmb);
5434                        ip_fanout_udp_conn(connp, mp1, (ipha_t *)mp1->b_rptr,
5435                            NULL, ira);
5436                        mutex_enter(&connfp->connf_lock);
5437                        /* Follow the next pointer before releasing the conn */
5438                        next_connp = connp->conn_next;
5439                        CONN_DEC_REF(connp);
5440                        connp = next_connp;
5441                }
5442        }

5444        /* Last one.  Send it upstream. */
5445        mutex_exit(&connfp->connf_lock);
5446        IP_STAT(ipst, ip_udp_fanmb);
5447        ip_fanout_udp_conn(connp, mp, ipha, NULL, ira);
5448        CONN_DEC_REF(connp);
5449 }

5451 /*
5452  * Split an incoming packet's IPv4 options into the label and the other options.
5453  * If 'allocate' is set it does memory allocation for the ip_pkt_t, including
5454  * clearing out any leftover label or options.
5455  * Otherwise it just makes ipp point into the packet.
5456  *
5457  * Returns zero if ok; ENOMEM if the buffer couldn't be allocated.
5458  */
5459 int
5460 ip_find_hdr_v4(ipha_t *ipha, ip_pkt_t *ipp, boolean_t allocate)
5461 {
5462        uchar_t         *opt;
5463        uint32_t        totallen;
5464        uint32_t        optval;
5465        uint32_t        optlen;

5467        ipp->ipp_fields |= IPPF_HOPLIMIT | IPPF_TCLASS | IPPF_ADDR;
5468        ipp->ipp_hoplimit = ipha->ipha_ttl;
5469        ipp->ipp_type_of_service = ipha->ipha_type_of_service;
5470        IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &ipp->ipp_addr);

5472        /*
5473         * Get length (in 4 byte octets) of IP header options.
```



```
5474         */
5475        totallen = ipha->ipha_version_and_hdr_length -
5476            (uint8_t)((IP_VERSION << 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);

5478        if (totallen == 0) {
5479                if (!allocate)
5480                        return (0);

5482                /* Clear out anything from a previous packet */
5483                if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
5484                        kmem_free(ipp->ipp_ipv4_options,
5485                            ipp->ipp_ipv4_options_len);
5486                        ipp->ipp_ipv4_options = NULL;
5487                        ipp->ipp_ipv4_options_len = 0;
5488                        ipp->ipp_fields &= ~IPPF_IPV4_OPTIONS;
5489                }
5490                if (ipp->ipp_fields & IPPF_LABEL_V4) {
5491                        kmem_free(ipp->ipp_label_v4, ipp->ipp_label_len_v4);
5492                        ipp->ipp_label_v4 = NULL;
5493                        ipp->ipp_label_len_v4 = 0;
5494                        ipp->ipp_fields &= ~IPPF_LABEL_V4;
5495                }
5496                return (0);
5497        }

5499        totallen <<= 2;
5500        opt = (uchar_t *)&ipha[1];
5501        if (!is_system_labeled()) {

5503        copyall:
5504                if (!allocate) {
5505                        if (totallen != 0) {
5506                                ipp->ipp_ipv4_options = opt;
5507                                ipp->ipp_ipv4_options_len = totallen;
5508                                ipp->ipp_fields |= IPPF_IPV4_OPTIONS;
5509                        }
5510                        return (0);
5511                }
5512                /* Just copy all of options */
5513                if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
5514                        if (totallen == ipp->ipp_ipv4_options_len) {
5515                                bcopy(opt, ipp->ipp_ipv4_options, totallen);
5516                                return (0);
5517                        }
5518                        kmem_free(ipp->ipp_ipv4_options,
5519                            ipp->ipp_ipv4_options_len);
5520                        ipp->ipp_ipv4_options = NULL;
5521                        ipp->ipp_ipv4_options_len = 0;
5522                        ipp->ipp_fields &= ~IPPF_IPV4_OPTIONS;
5523                }
5524                if (totallen == 0)
5525                        return (0);

5527                ipp->ipp_ipv4_options = kmem_alloc(totallen, KM_NOSLEEP);
5528                if (ipp->ipp_ipv4_options == NULL)
5529                        return (ENOMEM);
5530                ipp->ipp_ipv4_options_len = totallen;
5531                ipp->ipp_fields |= IPPF_IPV4_OPTIONS;
5532                bcopy(opt, ipp->ipp_ipv4_options, totallen);
5533                return (0);
5534        }

5536        if (allocate && (ipp->ipp_fields & IPPF_LABEL_V4)) {
5537                kmem_free(ipp->ipp_label_v4, ipp->ipp_label_len_v4);
5538                ipp->ipp_label_v4 = NULL;
5539                ipp->ipp_label_len_v4 = 0;
```

```
5540                ipp->ipp_fields &= ~IPPF_LABEL_V4;
5541        }

5543        /*
5544         * Search for CIPSO option.
5545         * We assume CIPSO is first in options if it is present.
5546         * If it isn't, then ipp_opt_ipv4_options will not include the options
5547         * prior to the CIPSO option.
5548         */
5549        while (totallen != 0) {
5550                switch (optval = opt[IPOPT_OPTVAL]) {
5551                case IPOPT_EOL:
5552                        return (0);
5553                case IPOPT_NOP:
5554                        optlen = 1;
5555                        break;
5556                default:
5557                        if (totallen <= IPOPT_OLEN)
5558                                return (EINVAL);
5559                        optlen = opt[IPOPT_OLEN];
5560                        if (optlen < 2)
5561                                return (EINVAL);
5562                }
5563                if (optlen > totallen)
5564                        return (EINVAL);

5566                switch (optval) {
5567                case IPOPT_COMSEC:
5568                        if (!allocate) {
5569                                ipp->ipp_label_v4 = opt;
5570                                ipp->ipp_label_len_v4 = optlen;
5571                                ipp->ipp_fields |= IPPF_LABEL_V4;
5572                        } else {
5573                                ipp->ipp_label_v4 = kmem_alloc(optlen,
5574                                    KM_NOSLEEP);
5575                                if (ipp->ipp_label_v4 == NULL)
5576                                        return (ENOMEM);
5577                                ipp->ipp_label_len_v4 = optlen;
5578                                ipp->ipp_fields |= IPPF_LABEL_V4;
5579                                bcopy(opt, ipp->ipp_label_v4, optlen);
5580                        }
5581                        totallen -= optlen;
5582                        opt += optlen;

5584                        /* Skip padding bytes until we get to a multiple of 4 */
5585                        while ((totallen & 3) != 0 && opt[0] == IPOPT_NOP) {
5586                                totallen--;
5587                                opt++;
5588                        }
5589                        /* Remaining as ipp_ipv4_options */
5590                        goto copyall;
5591                }
5592                totallen -= optlen;
5593                opt += optlen;
5594        }
5595        /* No CIPSO found; return everything as ipp_ipv4_options */
5596        totallen = ipha->ipha_version_and_hdr_length -
5597            (uint8_t)((IP_VERSION << 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);
5598        totallen <<= 2;
5599        opt = (uchar_t *)&ipha[1];
5600        goto copyall;
5601 }

5603 /*
5604  * Efficient versions of lookup for an IRE when we only
5605  * match the address.
```



```
5606  * For RTF_REJECT or BLACKHOLE we return IRE_NOROUTE.
5607  * Does not handle multicast addresses.
5608  */
5609 uint_t
5610 ip_type_v4(ipaddr_t addr, ip_stack_t *ipst)
5611 {
5612        ire_t *ire;
5613        uint_t result;

5615        ire = ire_ftable_lookup_simple_v4(addr, 0, ipst, NULL);
5616        ASSERT(ire != NULL);
5617        if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))
5618                result = IRE_NOROUTE;
5619        else
5620                result = ire->ire_type;
5621        ire_refrele(ire);
5622        return (result);
5623 }

5625 /*
5626  * Efficient versions of lookup for an IRE when we only
5627  * match the address.
5628  * For RTF_REJECT or BLACKHOLE we return IRE_NOROUTE.
5629  * Does not handle multicast addresses.
5630  */
5631 uint_t
5632 ip_type_v6(const in6_addr_t *addr, ip_stack_t *ipst)
5633 {
5634        ire_t *ire;
5635        uint_t result;

5637        ire = ire_ftable_lookup_simple_v6(addr, 0, ipst, NULL);
5638        ASSERT(ire != NULL);
5639        if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))
5640                result = IRE_NOROUTE;
5641        else
5642                result = ire->ire_type;
5643        ire_refrele(ire);
5644        return (result);
5645 }

5647 /*
5648  * Nobody should be sending
5649  * packets up this stream
5650  */
5651 static void
5652 ip_lrput(queue_t *q, mblk_t *mp)
5653 {
5654        switch (mp->b_datap->db_type) {
5655        case M_FLUSH:
5656                /* Turn around */
5657                if (*mp->b_rptr & FLUSHW) {
5658                        *mp->b_rptr &= ~FLUSHR;
5659                        qreply(q, mp);
5660                        return;
5661                }
5662                break;
5663        }
5664        freemsg(mp);
5665 }

5667 /* Nobody should be sending packets down this stream */
5668 /* ARGSUSED */
5669 void
5670 ip_lwput(queue_t *q, mblk_t *mp)
5671 {
```

```
5672         freemsg(mp);
5673 }

5675 /*
5676  * Move the first hop in any source route to ipha_dst and remove that part of
5677  * the source route.  Called by other protocols.  Errors in option formatting
5678  * are ignored - will be handled by ip_output_options. Return the final
5679  * destination (either ipha_dst or the last entry in a source route.)
5680  */
5681 ipaddr_t
5682 ip_massage_options(ipha_t *ipha, netstack_t *ns)
5683 {
5684         ipoptp_t        opts;
5685         uchar_t         *opt;
5686         uint8_t         optval;
5687         uint8_t         optlen;
5688         ipaddr_t        dst;
5689         int             i;
5690         ip_stack_t      *ipst = ns->netstack_ip;

5692         ip2dbg(("ip_massage_options\n"));
5693         dst = ipha->ipha_dst;
5694         for (optval = ipoptp_first(&opts, ipha);
5695             optval != IPOPT_EOL;
5696             optval = ipoptp_next(&opts)) {
5697                 opt = opts.ipoptp_cur;
5698                 switch (optval) {
5699                         uint8_t off;
5700                 case IPOPT_SSRR:
5701                 case IPOPT_LSRR:
5702                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
5703                                 ip1dbg(("ip_massage_options: bad src route\n"));
5704                                 break;
5705                         }
5706                         optlen = opts.ipoptp_len;
5707                         off = opt[IPOPT_OFFSET];
5708                         off--;
5709                 redo_srr:
5710                         if (optlen < IP_ADDR_LEN ||
5711                             off > optlen - IP_ADDR_LEN) {
5712                                 /* End of source route */
5713                                 ip1dbg(("ip_massage_options: end of SR\n"));
5714                                 break;
5715                         }
5716                         bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
5717                         ip1dbg(("ip_massage_options: next hop 0x%x\n",
5718                             ntohl(dst)));
5719                         /*
5720                          * Check if our address is present more than
5721                          * once as consecutive hops in source route.
5722                          * XXX verify per-interface ip_forwarding
5723                          * for source route?
5724                          */
5725                         if (ip_type_v4(dst, ipst) == IRE_LOCAL) {
5726                                 off += IP_ADDR_LEN;
5727                                 goto redo_srr;
5728                         }
5729                         if (dst == htonl(INADDR_LOOPBACK)) {
5730                                 ip1dbg(("ip_massage_options: loopback addr in "
5731                                     "source route!\n"));
5732                                 break;
5733                         }
5734                         /*
5735                          * Update ipha_dst to be the first hop and remove the
5736                          * first hop from the source route (by overwriting
5737                          * part of the option with NOP options).
```



```
5738                          */
5739                         ipha->ipha_dst = dst;
5740                         /* Put the last entry in dst */
5741                         off = ((optlen - IP_ADDR_LEN - 3) & ~(IP_ADDR_LEN-1)) +
5742                             3;
5743                         bcopy(&opt[off], &dst, IP_ADDR_LEN);

5745                         ip1dbg(("ip_massage_options: last hop 0x%x\n",
5746                             ntohl(dst)));
5747                         /* Move down and overwrite */
5748                         opt[IP_ADDR_LEN] = opt[0];
5749                         opt[IP_ADDR_LEN+1] = opt[IPOPT_OLEN] - IP_ADDR_LEN;
5750                         opt[IP_ADDR_LEN+2] = opt[IPOPT_OFFSET];
5751                         for (i = 0; i < IP_ADDR_LEN; i++)
5752                                 opt[i] = IPOPT_NOP;
5753                         break;
5754                 }
5755         }
5756         return (dst);
5757 }

5759 /*
5760  * Return the network mask
5761  * associated with the specified address.
5762  */
5763 ipaddr_t
5764 ip_net_mask(ipaddr_t addr)
5765 {
5766         uchar_t *up = (uchar_t *)&addr;
5767         ipaddr_t mask = 0;
5768         uchar_t *maskp = (uchar_t *)&mask;

5770 #if defined(__i386) || defined(__amd64)
5771 #define TOTALLY_BRAIN_DAMAGED_C_COMPILER
5772 #endif
5773 #ifdef  TOTALLY_BRAIN_DAMAGED_C_COMPILER
5774         maskp[0] = maskp[1] = maskp[2] = maskp[3] = 0;
5775 #endif
5776         if (CLASSD(addr)) {
5777                 maskp[0] = 0xF0;
5778                 return (mask);
5779         }

5781         /* We assume Class E default netmask to be 32 */
5782         if (CLASSE(addr))
5783                 return (0xffffffffU);

5785         if (addr == 0)
5786                 return (0);
5787         maskp[0] = 0xFF;
5788         if ((up[0] & 0x80) == 0)
5789                 return (mask);

5791         maskp[1] = 0xFF;
5792         if ((up[0] & 0xC0) == 0x80)
5793                 return (mask);

5795         maskp[2] = 0xFF;
5796         if ((up[0] & 0xE0) == 0xC0)
5797                 return (mask);

5799         /* Otherwise return no mask */
5800         return ((ipaddr_t)0);
5801 }

5803 /* Name/Value Table Lookup Routine */
```

```
5804 char *
5805 ip_nv_lookup(nv_t *nv, int value)
5806 {
5807         if (!nv)
5808                 return (NULL);
5809         for (; nv->nv_name; nv++) {
5810                 if (nv->nv_value == value)
5811                         return (nv->nv_name);
5812         }
5813         return ("unknown");
5814 }

5816 static int
5817 ip_wait_for_info_ack(ill_t *ill)
5818 {
5819         int err;

5821         mutex_enter(&ill->ill_lock);
5822         while (ill->ill_state_flags & ILL_LL_SUBNET_PENDING) {
5823                 /*
5824                  * Return value of 0 indicates a pending signal.
5825                  */
5826                 err = cv_wait_sig(&ill->ill_cv, &ill->ill_lock);
5827                 if (err == 0) {
5828                         mutex_exit(&ill->ill_lock);
5829                         return (EINTR);
5830                 }
5831         }
5832         mutex_exit(&ill->ill_lock);
5833         /*
5834          * ip_rput_other could have set an error  in ill_error on
5835          * receipt of M_ERROR.
5836          */
5837         return (ill->ill_error);
5838 }

5840 /*
5841  * This is a module open, i.e. this is a control stream for access
5842  * to a DLPI device.  We allocate an ill_t as the instance data in
5843  * this case.
5844  */
5845 static int
5846 ip_modopen(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
5847 {
5848         ill_t   *ill;
5849         int     err;
5850         zoneid_t zoneid;
5851         netstack_t *ns;
5852         ip_stack_t *ipst;

5854         /*
5855          * Prevent unprivileged processes from pushing IP so that
5856          * they can't send raw IP.
5857          */
5858         if (secpolicy_net_rawaccess(credp) != 0)
5859                 return (EPERM);

5861         ns = netstack_find_by_cred(credp);
5862         ASSERT(ns != NULL);
5863         ipst = ns->netstack_ip;
5864         ASSERT(ipst != NULL);

5866         /*
5867          * For exclusive stacks we set the zoneid to zero
5868          * to make IP operate as if in the global zone.
5869          */
```



```
5870         if (ipst->ips_netstack->netstack_stackid != GLOBAL_NETSTACKID)
5871                 zoneid = GLOBAL_ZONEID;
5872         else
5873                 zoneid = crgetzoneid(credp);

5875         ill = (ill_t *)mi_open_alloc_sleep(sizeof (ill_t));
5876         q->q_ptr = WR(q)->q_ptr = ill;
5877         ill->ill_ipst = ipst;
5878         ill->ill_zoneid = zoneid;

5880         /*
5881          * ill_init initializes the ill fields and then sends down
5882          * down a DL_INFO_REQ after calling qprocson.
5883          */
5884         err = ill_init(q, ill);

5886         if (err != 0) {
5887                 mi_free(ill);
5888                 netstack_rele(ipst->ips_netstack);
5889                 q->q_ptr = NULL;
5890                 WR(q)->q_ptr = NULL;
5891                 return (err);
5892         }

5894         /*
5895          * Wait for the DL_INFO_ACK if a DL_INFO_REQ was sent.
5896          *
5897          * ill_init initializes the ipsq marking this thread as
5898          * writer
5899          */
5900         ipsq_exit(ill->ill_phyint->phyint_ipsq);
5901         err = ip_wait_for_info_ack(ill);
5902         if (err == 0)
5903                 ill->ill_credp = credp;
5904         else
5905                 goto fail;

5907         crhold(credp);

5909         mutex_enter(&ipst->ips_ip_mi_lock);
5910         err = mi_open_link(&ipst->ips_ip_g_head, (IDP)q->q_ptr, devp, flag,
5911             sflag, credp);
5912         mutex_exit(&ipst->ips_ip_mi_lock);
5913 fail:
5914         if (err) {
5915                 (void) ip_close(q, 0);
5916                 return (err);
5917         }
5918         return (0);
5919 }

5921 /* For /dev/ip aka AF_INET open */
5922 int
5923 ip_openv4(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
5924 {
5925         return (ip_open(q, devp, flag, sflag, credp, B_FALSE));
5926 }

5928 /* For /dev/ip6 aka AF_INET6 open */
5929 int
5930 ip_openv6(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
5931 {
5932         return (ip_open(q, devp, flag, sflag, credp, B_TRUE));
5933 }

5935 /* IP open routine. */
```

```
5936 int
5937 ip_open(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp,
5938     boolean_t isv6)
5939 {
5940         conn_t          *connp;
5941         major_t         maj;
5942         zoneid_t        zoneid;
5943         netstack_t      *ns;
5944         ip_stack_t      *ipst;
5946         /* Allow reopen. */
5947         if (q->q_ptr != NULL)
5948                 return (0);
5950         if (sflag & MODOPEN) {
5951                 /* This is a module open */
5952                 return (ip_modopen(q, devp, flag, sflag, credp));
5953         }
5955         if ((flag & ~(FKLYR)) == IP_HELPER_STR) {
5956                 /*
5957                  * Non streams based socket looking for a stream
5958                  * to access IP
5959                  */
5960                 return (ip_helper_stream_setup(q, devp, flag, sflag,
5961                     credp, isv6));
5962         }
5964         ns = netstack_find_by_cred(credp);
5965         ASSERT(ns != NULL);
5966         ipst = ns->netstack_ip;
5967         ASSERT(ipst != NULL);
5969         /*
5970          * For exclusive stacks we set the zoneid to zero
5971          * to make IP operate as if in the global zone.
5972          */
5973         if (ipst->ips_netstack->netstack_stackid != GLOBAL_NETSTACKID)
5974                 zoneid = GLOBAL_ZONEID;
5975         else
5976                 zoneid = crgetzoneid(credp);
5978         /*
5979          * We are opening as a device. This is an IP client stream, and we
5980          * allocate an conn_t as the instance data.
5981          */
5982         connp = ipcl_conn_create(IPCL_IPCCONN, KM_SLEEP, ipst->ips_netstack);
5984         /*
5985          * ipcl_conn_create did a netstack_hold. Undo the hold that was
5986          * done by netstack_find_by_cred()
5987          */
5988         netstack_rele(ipst->ips_netstack);
5990         connp->conn_ixa->ixa_flags |= IXAF_MULTICAST_LOOP | IXAF_SET_ULP_CKSUM;
5991         /* conn_allzones can not be set this early, hence no IPCL_ZONEID */
5992         connp->conn_ixa->ixa_zoneid = zoneid;
5993         connp->conn_zoneid = zoneid;
5995         connp->conn_rq = q;
5996         q->q_ptr = WR(q)->q_ptr = connp;
5998         /* Minor tells us which /dev entry was opened */
5999         if (isv6) {
6000                 connp->conn_family = AF_INET6;
6001                 connp->conn_ipversion = IPV6_VERSION;
```



```
6002                 connp->conn_ixa->ixa_flags &= ~IXAF_IS_IPV4;
6003                 connp->conn_ixa->ixa_src_preferences = IPV6_PREFER_SRC_DEFAULT;
6004         } else {
6005                 connp->conn_family = AF_INET;
6006                 connp->conn_ipversion = IPV4_VERSION;
6007                 connp->conn_ixa->ixa_flags |= IXAF_IS_IPV4;
6008         }
6010         if ((ip_minor_arena_la != NULL) && (flag & SO_SOCKSTR) &&
6011             ((connp->conn_dev = inet_minor_alloc(ip_minor_arena_la)) != 0)) {
6012                 connp->conn_minor_arena = ip_minor_arena_la;
6013         } else {
6014                 /*
6015                  * Either minor numbers in the large arena were exhausted
6016                  * or a non socket application is doing the open.
6017                  * Try to allocate from the small arena.
6018                  */
6019                 if ((connp->conn_dev =
6020                     inet_minor_alloc(ip_minor_arena_sa)) == 0) {
6021                         /* CONN_DEC_REF takes care of netstack_rele() */
6022                         q->q_ptr = WR(q)->q_ptr = NULL;
6023                         CONN_DEC_REF(connp);
6024                         return (EBUSY);
6025                 }
6026                 connp->conn_minor_arena = ip_minor_arena_sa;
6027         }
6029         maj = getemajor(*devp);
6030         *devp = makedevice(maj, (minor_t)connp->conn_dev);
6032         /*
6033          * connp->conn_cred is crfree()ed in ipcl_conn_destroy()
6034          */
6035         connp->conn_cred = credp;
6036         connp->conn_cpid = curproc->p_pid;
6037         /* Cache things in ixa without an extra refhold */
6038         ASSERT(!(connp->conn_ixa->ixa_free_flags & IXA_FREE_CRED));
6039         connp->conn_ixa->ixa_cred = connp->conn_cred;
6040         connp->conn_ixa->ixa_cpid = connp->conn_cpid;
6041         if (is_system_labeled())
6042                 connp->conn_ixa->ixa_tsl = crgetlabel(connp->conn_cred);
6044         /*
6045          * Handle IP_IOC_RTS_REQUEST and other ioctls which use conn_recv
6046          */
6047         connp->conn_recv = ip_conn_input;
6048         connp->conn_recvicmp = ip_conn_input_icmp;
6050         crhold(connp->conn_cred);
6052         /*
6053          * If the caller has the process-wide flag set, then default to MAC
6054          * exempt mode.  This allows read-down to unlabeled hosts.
6055          */
6056         if (getpflags(NET_MAC_AWARE, credp) != 0)
6057                 connp->conn_mac_mode = CONN_MAC_AWARE;
6059         connp->conn_zone_is_global = (crgetzoneid(credp) == GLOBAL_ZONEID);
6061         connp->conn_rq = q;
6062         connp->conn_wq = WR(q);
6064         /* Non-zero default values */
6065         connp->conn_ixa->ixa_flags |= IXAF_MULTICAST_LOOP;
6067         /*
```

```
6068          * Make the conn globally visible to walkers
6069          */
6070         ASSERT(connp->conn_ref == 1);
6071         mutex_enter(&connp->conn_lock);
6072         connp->conn_state_flags &= ~CONN_INCIPIENT;
6073         mutex_exit(&connp->conn_lock);

6075         qprocson(q);

6077         return (0);
6078 }

6080 /*
6081  * Set IPsec policy from an ipsec_req_t. If the req is not "zero" and valid,
6082  * all of them are copied to the conn_t. If the req is "zero", the policy is
6083  * zeroed out. A "zero" policy has zero ipsr_{ah,req,self_encap}_req
6084  * fields.
6085  * We keep only the latest setting of the policy and thus policy setting
6086  * is not incremental/cumulative.
6087  *
6088  * Requests to set policies with multiple alternative actions will
6089  * go through a different API.
6090  */
6091 int
6092 ipsec_set_req(cred_t *cr, conn_t *connp, ipsec_req_t *req)
6093 {
6094         uint_t ah_req = 0;
6095         uint_t esp_req = 0;
6096         uint_t se_req = 0;
6097         ipsec_act_t *actp = NULL;
6098         uint_t nact;
6099         ipsec_policy_head_t *ph;
6100         boolean_t is_pol_reset, is_pol_inserted = B_FALSE;
6101         int error = 0;
6102         netstack_t      *ns = connp->conn_netstack;
6103         ip_stack_t      *ipst = ns->netstack_ip;
6104         ipsec_stack_t   *ipss = ns->netstack_ipsec;

6106 #define REQ_MASK (IPSEC_PREF_REQUIRED|IPSEC_PREF_NEVER)

6108         /*
6109          * The IP_SEC_OPT option does not allow variable length parameters,
6110          * hence a request cannot be NULL.
6111          */
6112         if (req == NULL)
6113                 return (EINVAL);

6115         ah_req = req->ipsr_ah_req;
6116         esp_req = req->ipsr_esp_req;
6117         se_req = req->ipsr_self_encap_req;

6119         /* Don't allow setting self-encap without one or more of AH/ESP. */
6120         if (se_req != 0 && esp_req == 0 && ah_req == 0)
6121                 return (EINVAL);

6123         /*
6124          * Are we dealing with a request to reset the policy (i.e.
6125          * zero requests).
6126          */
6127         is_pol_reset = ((ah_req & REQ_MASK) == 0 &&
6128             (esp_req & REQ_MASK) == 0 &&
6129             (se_req & REQ_MASK) == 0);

6131         if (!is_pol_reset) {
6132                 /*
6133                  * If we couldn't load IPsec, fail with "protocol
```



```
6134                  * not supported".
6135                  * IPsec may not have been loaded for a request with zero
6136                  * policies, so we don't fail in this case.
6137                  */
6138                 mutex_enter(&ipss->ipsec_loader_lock);
6139                 if (ipss->ipsec_loader_state != IPSEC_LOADER_SUCCEEDED) {
6140                         mutex_exit(&ipss->ipsec_loader_lock);
6141                         return (EPROTONOSUPPORT);
6142                 }
6143                 mutex_exit(&ipss->ipsec_loader_lock);

6145                 /*
6146                  * Test for valid requests. Invalid algorithms
6147                  * need to be tested by IPsec code because new
6148                  * algorithms can be added dynamically.
6149                  */
6150                 if ((ah_req & ~(REQ_MASK|IPSEC_PREF_UNIQUE)) != 0 ||
6151                     (esp_req & ~(REQ_MASK|IPSEC_PREF_UNIQUE)) != 0 ||
6152                     (se_req & ~(REQ_MASK|IPSEC_PREF_UNIQUE)) != 0) {
6153                         return (EINVAL);
6154                 }

6156                 /*
6157                  * Only privileged users can issue these
6158                  * requests.
6159                  */
6160                 if (((ah_req & IPSEC_PREF_NEVER) ||
6161                     (esp_req & IPSEC_PREF_NEVER) ||
6162                     (se_req & IPSEC_PREF_NEVER)) &&
6163                     secpolicy_ip_config(cr, B_FALSE) != 0) {
6164                         return (EPERM);
6165                 }

6167                 /*
6168                  * The IPSEC_PREF_REQUIRED and IPSEC_PREF_NEVER
6169                  * are mutually exclusive.
6170                  */
6171                 if (((ah_req & REQ_MASK) == REQ_MASK) ||
6172                     ((esp_req & REQ_MASK) == REQ_MASK) ||
6173                     ((se_req & REQ_MASK) == REQ_MASK)) {
6174                         /* Both of them are set */
6175                         return (EINVAL);
6176                 }
6177         }

6179         ASSERT(MUTEX_HELD(&connp->conn_lock));

6181         /*
6182          * If we have already cached policies in conn_connect(), don't
6183          * let them change now. We cache policies for connections
6184          * whose src,dst [addr, port] is known.
6185          */
6186         if (connp->conn_policy_cached) {
6187                 return (EINVAL);
6188         }

6190         /*
6191          * We have a zero policies, reset the connection policy if already
6192          * set. This will cause the connection to inherit the
6193          * global policy, if any.
6194          */
6195         if (is_pol_reset) {
6196                 if (connp->conn_policy != NULL) {
6197                         IPPH_REFRELE(connp->conn_policy, ipst->ips_netstack);
6198                         connp->conn_policy = NULL;
6199                 }
```

```
6200                connp->conn_in_enforce_policy = B_FALSE;
6201                connp->conn_out_enforce_policy = B_FALSE;
6202                return (0);
6203        }

6205        ph = connp->conn_policy = ipsec_polhead_split(connp->conn_policy,
6206            ipst->ips_netstack);
6207        if (ph == NULL)
6208                goto enomem;

6210        ipsec_actvec_from_req(req, &actp, &nact, ipst->ips_netstack);
6211        if (actp == NULL)
6212                goto enomem;

6214        /*
6215         * Always insert IPv4 policy entries, since they can also apply to
6216         * ipv6 sockets being used in ipv4-compat mode.
6217         */
6218        if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V4,
6219            IPSEC_TYPE_INBOUND, ns))
6220                goto enomem;
6221        is_pol_inserted = B_TRUE;
6222        if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V4,
6223            IPSEC_TYPE_OUTBOUND, ns))
6224                goto enomem;

6226        /*
6227         * We're looking at a v6 socket, also insert the v6-specific
6228         * entries.
6229         */
6230        if (connp->conn_family == AF_INET6) {
6231                if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V6,
6232                    IPSEC_TYPE_INBOUND, ns))
6233                        goto enomem;
6234                if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V6,
6235                    IPSEC_TYPE_OUTBOUND, ns))
6236                        goto enomem;
6237        }

6239        ipsec_actvec_free(actp, nact);

6241        /*
6242         * If the requests need security, set enforce_policy.
6243         * If the requests are IPSEC_PREF_NEVER, one should
6244         * still set conn_out_enforce_policy so that ip_set_destination
6245         * marks the ip_xmit_attr_t appropriatly. This is needed so that
6246         * for connections that we don't cache policy in at connect time,
6247         * if global policy matches in ip_output_attach_policy, we
6248         * don't wrongly inherit global policy. Similarly, we need
6249         * to set conn_in_enforce_policy also so that we don't verify
6250         * policy wrongly.
6251         */
6252        if ((ah_req & REQ_MASK) != 0 ||
6253            (esp_req & REQ_MASK) != 0 ||
6254            (se_req & REQ_MASK) != 0) {
6255                connp->conn_in_enforce_policy = B_TRUE;
6256                connp->conn_out_enforce_policy = B_TRUE;
6257        }

6259        return (error);
6260 #undef REQ_MASK

6262        /*
6263         * Common memory-allocation-failure exit path.
6264         */
6265 enomem:
```



```
6266        if (actp != NULL)
6267                ipsec_actvec_free(actp, nact);
6268        if (is_pol_inserted)
6269                ipsec_polhead_flush(ph, ns);
6270        return (ENOMEM);
6271 }

6273 /*
6274  * Set socket options for joining and leaving multicast groups.
6275  * Common to IPv4 and IPv6; inet6 indicates the type of socket.
6276  * The caller has already check that the option name is consistent with
6277  * the address family of the socket.
6278  */
6279 int
6280 ip_opt_set_multicast_group(conn_t *connp, t_scalar_t name,
6281     uchar_t *invalp, boolean_t inet6, boolean_t checkonly)
6282 {
6283        int             *i1 = (int *)invalp;
6284        int             error = 0;
6285        ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
6286        struct ip_mreq  *v4_mreqp;
6287        struct ipv6_mreq *v6_mreqp;
6288        struct group_req *greqp;
6289        ire_t *ire;
6290        boolean_t done = B_FALSE;
6291        ipaddr_t ifaddr;
6292        in6_addr_t v6group;
6293        uint_t ifindex;
6294        boolean_t mcast_opt = B_TRUE;
6295        mcast_record_t fmode;
6296        int (*optfn)(conn_t *, boolean_t, const in6_addr_t *,
6297            ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *);

6299        switch (name) {
6300        case IP_ADD_MEMBERSHIP:
6301        case IPV6_JOIN_GROUP:
6302                mcast_opt = B_FALSE;
6303                /* FALLTHRU */
6304        case MCAST_JOIN_GROUP:
6305                fmode = MODE_IS_EXCLUDE;
6306                optfn = ip_opt_add_group;
6307                break;

6309        case IP_DROP_MEMBERSHIP:
6310        case IPV6_LEAVE_GROUP:
6311                mcast_opt = B_FALSE;
6312                /* FALLTHRU */
6313        case MCAST_LEAVE_GROUP:
6314                fmode = MODE_IS_INCLUDE;
6315                optfn = ip_opt_delete_group;
6316                break;
6317        default:
6318                ASSERT(0);
6319        }

6321        if (mcast_opt) {
6322                struct sockaddr_in *sin;
6323                struct sockaddr_in6 *sin6;

6325                greqp = (struct group_req *)i1;
6326                if (greqp->gr_group.ss_family == AF_INET) {
6327                        sin = (struct sockaddr_in *)&(greqp->gr_group);
6328                        IN6_INADDR_TO_V4MAPPED(&sin->sin_addr, &v6group);
6329                } else {
6330                        if (!inet6)
6331                                return (EINVAL);        /* Not on INET socket */
```

```
6333                        sin6 = (struct sockaddr_in6 *)&(greqp->gr_group);
6334                        v6group = sin6->sin6_addr;
6335                }
6336                ifaddr = INADDR_ANY;
6337                ifindex = greqp->gr_interface;
6338        } else if (inet6) {
6339                v6_mreqp = (struct ipv6_mreq *)i1;
6340                v6group = v6_mreqp->ipv6mr_multiaddr;
6341                ifaddr = INADDR_ANY;
6342                ifindex = v6_mreqp->ipv6mr_interface;
6343        } else {
6344                v4_mreqp = (struct ip_mreq *)i1;
6345                IN6_INADDR_TO_V4MAPPED(&v4_mreqp->imr_multiaddr, &v6group);
6346                ifaddr = (ipaddr_t)v4_mreqp->imr_interface.s_addr;
6347                ifindex = 0;
6348        }

6350        /*
6351         * In the multirouting case, we need to replicate
6352         * the request on all interfaces that will take part
6353         * in replication.  We do so because multirouting is
6354         * reflective, thus we will probably receive multi-
6355         * casts on those interfaces.
6356         * The ip_multirt_apply_membership() succeeds if
6357         * the operation succeeds on at least one interface.
6358         */
6359        if (IN6_IS_ADDR_V4MAPPED(&v6group)) {
6360                ipaddr_t group;

6362                IN6_V4MAPPED_TO_IPADDR(&v6group, group);

6364                ire = ire_ftable_lookup_v4(group, IP_HOST_MASK, 0,
6365                    IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6366                    MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6367        } else {
6368                ire = ire_ftable_lookup_v6(&v6group, &ipv6_all_ones, 0,
6369                    IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6370                    MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6371        }
6372        if (ire != NULL) {
6373                if (ire->ire_flags & RTF_MULTIRT) {
6374                        error = ip_multirt_apply_membership(optfn, ire, connp,
6375                            checkonly, &v6group, fmode, &ipv6_all_zeros);
6376                        done = B_TRUE;
6377                }
6378                ire_refrele(ire);
6379        }

6381        if (!done) {
6382                error = optfn(connp, checkonly, &v6group, ifaddr, ifindex,
6383                    fmode, &ipv6_all_zeros);
6384        }
6385        return (error);
6386 }

6388 /*
6389  * Set socket options for joining and leaving multicast groups
6390  * for specific sources.
6391  * Common to IPv4 and IPv6; inet6 indicates the type of socket.
6392  * The caller has already check that the option name is consistent with
6393  * the address family of the socket.
6394  */
6395 int
6396 ip_opt_set_multicast_sources(conn_t *connp, t_scalar_t name,
6397     uchar_t *invalp, boolean_t inet6, boolean_t checkonly)
```



```
6398 {
6399        int             *i1 = (int *)invalp;
6400        int             error = 0;
6401        ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
6402        struct ip_mreq_source *imreqp;
6403        struct group_source_req *gsreqp;
6404        in6_addr_t v6group, v6src;
6405        uint32_t ifindex;
6406        ipaddr_t ifaddr;
6407        boolean_t mcast_opt = B_TRUE;
6408        mcast_record_t fmode;
6409        ire_t *ire;
6410        boolean_t done = B_FALSE;
6411        int (*optfn)(conn_t *, boolean_t, const in6_addr_t *,
6412            ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *);

6414        switch (name) {
6415        case IP_BLOCK_SOURCE:
6416                mcast_opt = B_FALSE;
6417                /* FALLTHRU */
6418        case MCAST_BLOCK_SOURCE:
6419                fmode = MODE_IS_EXCLUDE;
6420                optfn = ip_opt_add_group;
6421                break;

6423        case IP_UNBLOCK_SOURCE:
6424                mcast_opt = B_FALSE;
6425                /* FALLTHRU */
6426        case MCAST_UNBLOCK_SOURCE:
6427                fmode = MODE_IS_EXCLUDE;
6428                optfn = ip_opt_delete_group;
6429                break;

6431        case IP_ADD_SOURCE_MEMBERSHIP:
6432                mcast_opt = B_FALSE;
6433                /* FALLTHRU */
6434        case MCAST_JOIN_SOURCE_GROUP:
6435                fmode = MODE_IS_INCLUDE;
6436                optfn = ip_opt_add_group;
6437                break;

6439        case IP_DROP_SOURCE_MEMBERSHIP:
6440                mcast_opt = B_FALSE;
6441                /* FALLTHRU */
6442        case MCAST_LEAVE_SOURCE_GROUP:
6443                fmode = MODE_IS_INCLUDE;
6444                optfn = ip_opt_delete_group;
6445                break;
6446        default:
6447                ASSERT(0);
6448        }

6450        if (mcast_opt) {
6451                gsreqp = (struct group_source_req *)i1;
6452                ifindex = gsreqp->gsr_interface;
6453                if (gsreqp->gsr_group.ss_family == AF_INET) {
6454                        struct sockaddr_in *s;
6455                        s = (struct sockaddr_in *)&gsreqp->gsr_group;
6456                        IN6_INADDR_TO_V4MAPPED(&s->sin_addr, &v6group);
6457                        s = (struct sockaddr_in *)&gsreqp->gsr_source;
6458                        IN6_INADDR_TO_V4MAPPED(&s->sin_addr, &v6src);
6459                } else {
6460                        struct sockaddr_in6 *s6;

6462                        if (!inet6)
6463                                return (EINVAL);        /* Not on INET socket */
```

```
6465                        s6 = (struct sockaddr_in6 *)&gsreqp->gsr_group;
6466                        v6group = s6->sin6_addr;
6467                        s6 = (struct sockaddr_in6 *)&gsreqp->gsr_source;
6468                        v6src = s6->sin6_addr;
6469                }
6470                ifaddr = INADDR_ANY;
6471        } else {
6472                imreqp = (struct ip_mreq_source *)i1;
6473                IN6_INADDR_TO_V4MAPPED(&imreqp->imr_multiaddr, &v6group);
6474                IN6_INADDR_TO_V4MAPPED(&imreqp->imr_sourceaddr, &v6src);
6475                ifaddr = (ipaddr_t)imreqp->imr_interface.s_addr;
6476                ifindex = 0;
6477        }

6479        /*
6480         * Handle src being mapped INADDR_ANY by changing it to unspecified.
6481         */
6482        if (IN6_IS_ADDR_V4MAPPED_ANY(&v6src))
6483                v6src = ipv6_all_zeros;

6485        /*
6486         * In the multirouting case, we need to replicate
6487         * the request as noted in the mcast cases above.
6488         */
6489        if (IN6_IS_ADDR_V4MAPPED(&v6group)) {
6490                ipaddr_t group;

6492                IN6_V4MAPPED_TO_IPADDR(&v6group, group);

6494                ire = ire_ftable_lookup_v4(group, IP_HOST_MASK, 0,
6495                    IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6496                    MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6497        } else {
6498                ire = ire_ftable_lookup_v6(&v6group, &ipv6_all_ones, 0,
6499                    IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6500                    MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6501        }
6502        if (ire != NULL) {
6503                if (ire->ire_flags & RTF_MULTIRT) {
6504                        error = ip_multirt_apply_membership(optfn, ire, connp,
6505                            checkonly, &v6group, fmode, &v6src);
6506                        done = B_TRUE;
6507                }
6508                ire_refrele(ire);
6509        }
6510        if (!done) {
6511                error = optfn(connp, checkonly, &v6group, ifaddr, ifindex,
6512                    fmode, &v6src);
6513        }
6514        return (error);
6515 }

6517 /*
6518  * Given a destination address and a pointer to where to put the information
6519  * this routine fills in the mtuinfo.
6520  * The socket must be connected.
6521  * For sctp conn_faddr is the primary address.
6522  */
6523 int
6524 ip_fill_mtuinfo(conn_t *connp, ip_xmit_attr_t *ixa, struct ip6_mtuinfo *mtuinfo)
6525 {
6526        uint32_t        pmtu = IP_MAXPACKET;
6527        uint_t          scopeid;

6529        if (IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6))
```



```
6530                return (-1);

6532        /* In case we never sent or called ip_set_destination_v4/v6 */
6533        if (ixa->ixa_ire != NULL)
6534                pmtu = ip_get_pmtu(ixa);

6536        if (ixa->ixa_flags & IXAF_SCOPEID_SET)
6537                scopeid = ixa->ixa_scopeid;
6538        else
6539                scopeid = 0;

6541        bzero(mtuinfo, sizeof (*mtuinfo));
6542        mtuinfo->ip6m_addr.sin6_family = AF_INET6;
6543        mtuinfo->ip6m_addr.sin6_port = connp->conn_fport;
6544        mtuinfo->ip6m_addr.sin6_addr = connp->conn_faddr_v6;
6545        mtuinfo->ip6m_addr.sin6_scope_id = scopeid;
6546        mtuinfo->ip6m_mtu = pmtu;

6548        return (sizeof (struct ip6_mtuinfo));
6549 }

6551 /*
6552  * When the src multihoming is changed from weak to [strong, preferred]
6553  * ip_ire_rebind_walker is called to walk the list of all ire_t entries
6554  * and identify routes that were created by user-applications in the
6555  * unbound state (i.e., without RTA_IFP), and for which an ire_ill is not
6556  * currently defined. These routes are then 'rebound', i.e., their ire_ill
6557  * is selected by finding an interface route for the gateway.
6558  */
6559 /* ARGSUSED */
6560 void
6561 ip_ire_rebind_walker(ire_t *ire, void *notused)
6562 {
6563        if (!ire->ire_unbound || ire->ire_ill != NULL)
6564                return;
6565        ire_rebind(ire);
6566        ire_delete(ire);
6567 }

6569 /*
6570  * When the src multihoming is changed from  [strong, preferred] to weak,
6571  * ip_ire_unbind_walker is called to walk the list of all ire_t entries, and
6572  * set any entries that were created by user-applications in the unbound state
6573  * (i.e., without RTA_IFP) back to having a NULL ire_ill.
6574  */
6575 /* ARGSUSED */
6576 void
6577 ip_ire_unbind_walker(ire_t *ire, void *notused)
6578 {
6579        ire_t *new_ire;

6581        if (!ire->ire_unbound || ire->ire_ill == NULL)
6582                return;
6583        if (ire->ire_ipversion == IPV6_VERSION) {
6584                new_ire = ire_create_v6(&ire->ire_addr_v6, &ire->ire_mask_v6,
6585                    &ire->ire_gateway_addr_v6, ire->ire_type, NULL,
6586                    ire->ire_zoneid, ire->ire_flags, NULL, ire->ire_ipst);
6587        } else {
6588                new_ire = ire_create((uchar_t *)&ire->ire_addr,
6589                    (uchar_t *)&ire->ire_mask,
6590                    (uchar_t *)&ire->ire_gateway_addr, ire->ire_type, NULL,
6591                    ire->ire_zoneid, ire->ire_flags, NULL, ire->ire_ipst);
6592        }
6593        if (new_ire == NULL)
6594                return;
6595        new_ire->ire_unbound = B_TRUE;
```

```
6596         /*
6597          * The bound ire must first be deleted so that we don't return
6598          * the existing one on the attempt to add the unbound new_ire.
6599          */
6600         ire_delete(ire);
6601         new_ire = ire_add(new_ire);
6602         if (new_ire != NULL)
6603                 ire_refrele(new_ire);
6604 }

6606 /*
6607  * When the settings of ip*_strict_src_multihoming tunables are changed,
6608  * all cached routes need to be recomputed. This recomputation needs to be
6609  * done when going from weaker to stronger modes so that the cached ire
6610  * for the connection does not violate the current ip*_strict_src_multihoming
6611  * setting. It also needs to be done when going from stronger to weaker modes,
6612  * so that we fall back to matching on the longest-matching-route (as opposed
6613  * to a shorter match that may have been selected in the strong mode
6614  * to satisfy src_multihoming settings).
6615  *
6616  * The cached ixa_ire entires for all conn_t entries are marked as
6617  * "verify" so that they will be recomputed for the next packet.
6618  */
6619 void
6620 conn_ire_revalidate(conn_t *connp, void *arg)
6621 {
6622         boolean_t isv6 = (boolean_t)arg;

6624         if ((isv6 && connp->conn_ipversion != IPV6_VERSION) ||
6625             (!isv6 && connp->conn_ipversion != IPV4_VERSION))
6626                 return;
6627         connp->conn_ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
6628 }

6630 /*
6631  * Handles both IPv4 and IPv6 reassembly - doing the out-of-order cases,
6632  * When an ipf is passed here for the first time, if
6633  * we already have in-order fragments on the queue, we convert from the fast-
6634  * path reassembly scheme to the hard-case scheme.  From then on, additional
6635  * fragments are reassembled here.  We keep track of the start and end offsets
6636  * of each piece, and the number of holes in the chain.  When the hole count
6637  * goes to zero, we are done!
6638  *
6639  * The ipf_count will be updated to account for any mblk(s) added (pointed to
6640  * by mp) or subtracted (freeb()ed dups), upon return the caller must update
6641  * ipfb_count and ill_frag_count by the difference of ipf_count before and
6642  * after the call to ip_reassemble().
6643  */
6644 int
6645 ip_reassemble(mblk_t *mp, ipf_t *ipf, uint_t start, boolean_t more, ill_t *ill,
6646     size_t msg_len)
6647 {
6648         uint_t  end;
6649         mblk_t  *next_mp;
6650         mblk_t  *mp1;
6651         uint_t  offset;
6652         boolean_t incr_dups = B_TRUE;
6653         boolean_t offset_zero_seen = B_FALSE;
6654         boolean_t pkt_boundary_checked = B_FALSE;

6656         /* If start == 0 then ipf_nf_hdr_len has to be set. */
6657         ASSERT(start != 0 || ipf->ipf_nf_hdr_len != 0);

6659         /* Add in byte count */
6660         ipf->ipf_count += msg_len;
6661         if (ipf->ipf_end) {
```



```
6662                 /*
6663                  * We were part way through in-order reassembly, but now there
6664                  * is a hole.  We walk through messages already queued, and
6665                  * mark them for hard case reassembly.  We know that up till
6666                  * now they were in order starting from offset zero.
6667                  */
6668                 offset = 0;
6669                 for (mp1 = ipf->ipf_mp->b_cont; mp1; mp1 = mp1->b_cont) {
6670                         IP_REASS_SET_START(mp1, offset);
6671                         if (offset == 0) {
6672                                 ASSERT(ipf->ipf_nf_hdr_len != 0);
6673                                 offset = -ipf->ipf_nf_hdr_len;
6674                         }
6675                         offset += mp1->b_wptr - mp1->b_rptr;
6676                         IP_REASS_SET_END(mp1, offset);
6677                 }
6678                 /* One hole at the end. */
6679                 ipf->ipf_hole_cnt = 1;
6680                 /* Brand it as a hard case, forever. */
6681                 ipf->ipf_end = 0;
6682         }
6683         /* Walk through all the new pieces. */
6684         do {
6685                 end = start + (mp->b_wptr - mp->b_rptr);
6686                 /*
6687                  * If start is 0, decrease 'end' only for the first mblk of
6688                  * the fragment. Otherwise 'end' can get wrong value in the
6689                  * second pass of the loop if first mblk is exactly the
6690                  * size of ipf_nf_hdr_len.
6691                  */
6692                 if (start == 0 && !offset_zero_seen) {
6693                         /* First segment */
6694                         ASSERT(ipf->ipf_nf_hdr_len != 0);
6695                         end -= ipf->ipf_nf_hdr_len;
6696                         offset_zero_seen = B_TRUE;
6697                 }
6698                 next_mp = mp->b_cont;
6699                 /*
6700                  * We are checking to see if there is any interesing data
6701                  * to process.  If there isn't and the mblk isn't the
6702                  * one which carries the unfragmentable header then we
6703                  * drop it.  It's possible to have just the unfragmentable
6704                  * header come through without any data.  That needs to be
6705                  * saved.
6706                  *
6707                  * If the assert at the top of this function holds then the
6708                  * term "ipf->ipf_nf_hdr_len != 0" isn't needed.  This code
6709                  * is infrequently traveled enough that the test is left in
6710                  * to protect against future code changes which break that
6711                  * invariant.
6712                  */
6713                 if (start == end && start != 0 && ipf->ipf_nf_hdr_len != 0) {
6714                         /* Empty.  Blast it. */
6715                         IP_REASS_SET_START(mp, 0);
6716                         IP_REASS_SET_END(mp, 0);
6717                         /*
6718                          * If the ipf points to the mblk we are about to free,
6719                          * update ipf to point to the next mblk (or NULL
6720                          * if none).
6721                          */
6722                         if (ipf->ipf_mp->b_cont == mp)
6723                                 ipf->ipf_mp->b_cont = next_mp;
6724                         freeb(mp);
6725                         continue;
6726                 }
6727                 mp->b_cont = NULL;
```

```
6728                IP_REASS_SET_START(mp, start);
6729                IP_REASS_SET_END(mp, end);
6730                if (!ipf->ipf_tail_mp) {
6731                        ipf->ipf_tail_mp = mp;
6732                        ipf->ipf_mp->b_cont = mp;
6733                        if (start == 0 || !more) {
6734                                ipf->ipf_hole_cnt = 1;
6735                                /*
6736                                 * if the first fragment comes in more than one
6737                                 * mblk, this loop will be executed for each
6738                                 * mblk. Need to adjust hole count so exiting
6739                                 * this routine will leave hole count at 1.
6740                                 */
6741                                if (next_mp)
6742                                        ipf->ipf_hole_cnt++;
6743                        } else
6744                                ipf->ipf_hole_cnt = 2;
6745                        continue;
6746                } else if (ipf->ipf_last_frag_seen && !more &&
6747                    !pkt_boundary_checked) {
6748                        /*
6749                         * We check datagram boundary only if this fragment
6750                         * claims to be the last fragment and we have seen a
6751                         * last fragment in the past too. We do this only
6752                         * once for a given fragment.
6753                         *
6754                         * start cannot be 0 here as fragments with start=0
6755                         * and MF=0 gets handled as a complete packet. These
6756                         * fragments should not reach here.
6757                         */
6759                        if (start + msgdsize(mp) !=
6760                            IP_REASS_END(ipf->ipf_tail_mp)) {
6761                                /*
6762                                 * We have two fragments both of which claim
6763                                 * to be the last fragment but gives conflicting
6764                                 * information about the whole datagram size.
6765                                 * Something fishy is going on. Drop the
6766                                 * fragment and free up the reassembly list.
6767                                 */
6768                                return (IP_REASS_FAILED);
6769                        }
6771                        /*
6772                         * We shouldn't come to this code block again for this
6773                         * particular fragment.
6774                         */
6775                        pkt_boundary_checked = B_TRUE;
6776                }
6778                /* New stuff at or beyond tail? */
6779                offset = IP_REASS_END(ipf->ipf_tail_mp);
6780                if (start >= offset) {
6781                        if (ipf->ipf_last_frag_seen) {
6782                                /* current fragment is beyond last fragment */
6783                                return (IP_REASS_FAILED);
6784                        }
6785                        /* Link it on end. */
6786                        ipf->ipf_tail_mp->b_cont = mp;
6787                        ipf->ipf_tail_mp = mp;
6788                        if (more) {
6789                                if (start != offset)
6790                                        ipf->ipf_hole_cnt++;
6791                        } else if (start == offset && next_mp == NULL)
6792                                        ipf->ipf_hole_cnt--;
6793                        continue;
```



```
6794                }
6795                mp1 = ipf->ipf_mp->b_cont;
6796                offset = IP_REASS_START(mp1);
6797                /* New stuff at the front? */
6798                if (start < offset) {
6799                        if (start == 0) {
6800                                if (end >= offset) {
6801                                        /* Nailed the hole at the begining. */
6802                                        ipf->ipf_hole_cnt--;
6803                                }
6804                        } else if (end < offset) {
6805                                /*
6806                                 * A hole, stuff, and a hole where there used
6807                                 * to be just a hole.
6808                                 */
6809                                ipf->ipf_hole_cnt++;
6810                        }
6811                        mp->b_cont = mp1;
6812                        /* Check for overlap. */
6813                        while (end > offset) {
6814                                if (end < IP_REASS_END(mp1)) {
6815                                        mp->b_wptr -= end - offset;
6816                                        IP_REASS_SET_END(mp, offset);
6817                                        BUMP_MIB(ill->ill_ip_mib,
6818                                            ipIfStatsReasmPartDups);
6819                                        break;
6820                                }
6821                                /* Did we cover another hole? */
6822                                if ((mp1->b_cont &&
6823                                    IP_REASS_END(mp1) !=
6824                                    IP_REASS_START(mp1->b_cont) &&
6825                                    end >= IP_REASS_START(mp1->b_cont)) ||
6826                                    (!ipf->ipf_last_frag_seen && !more)) {
6827                                        ipf->ipf_hole_cnt--;
6828                                }
6829                                /* Clip out mp1. */
6830                                if ((mp->b_cont = mp1->b_cont) == NULL) {
6831                                        /*
6832                                         * After clipping out mp1, this guy
6833                                         * is now hanging off the end.
6834                                         */
6835                                        ipf->ipf_tail_mp = mp;
6836                                }
6837                                IP_REASS_SET_START(mp1, 0);
6838                                IP_REASS_SET_END(mp1, 0);
6839                                /* Subtract byte count */
6840                                ipf->ipf_count -= mp1->b_datap->db_lim -
6841                                    mp1->b_datap->db_base;
6842                                freeb(mp1);
6843                                BUMP_MIB(ill->ill_ip_mib,
6844                                    ipIfStatsReasmPartDups);
6845                                mp1 = mp->b_cont;
6846                                if (!mp1)
6847                                        break;
6848                                offset = IP_REASS_START(mp1);
6849                        }
6850                        ipf->ipf_mp->b_cont = mp;
6851                        continue;
6852                }
6853                /*
6854                 * The new piece starts somewhere between the start of the head
6855                 * and before the end of the tail.
6856                 */
6857                for (; mp1; mp1 = mp1->b_cont) {
6858                        offset = IP_REASS_END(mp1);
6859                        if (start < offset) {
```

```
6860                                if (end <= offset) {
6861                                        /* Nothing new. */
6862                                        IP_REASS_SET_START(mp, 0);
6863                                        IP_REASS_SET_END(mp, 0);
6864                                        /* Subtract byte count */
6865                                        ipf->ipf_count -= mp->b_datap->db_lim -
6866                                            mp->b_datap->db_base;
6867                                        if (incr_dups) {
6868                                                ipf->ipf_num_dups++;
6869                                                incr_dups = B_FALSE;
6870                                        }
6871                                        freeb(mp);
6872                                        BUMP_MIB(ill->ill_ip_mib,
6873                                            ipIfStatsReasmDuplicates);
6874                                        break;
6875                                }
6876                                /*
6877                                 * Trim redundant stuff off beginning of new
6878                                 * piece.
6879                                 */
6880                                IP_REASS_SET_START(mp, offset);
6881                                mp->b_rptr += offset - start;
6882                                BUMP_MIB(ill->ill_ip_mib,
6883                                    ipIfStatsReasmPartDups);
6884                                start = offset;
6885                                if (!mp1->b_cont) {
6886                                        /*
6887                                         * After trimming, this guy is now
6888                                         * hanging off the end.
6889                                         */
6890                                        mp1->b_cont = mp;
6891                                        ipf->ipf_tail_mp = mp;
6892                                        if (!more) {
6893                                                ipf->ipf_hole_cnt--;
6894                                        }
6895                                        break;
6896                                }
6897                        }
6898                        if (start >= IP_REASS_START(mp1->b_cont))
6899                                continue;
6900                        /* Fill a hole */
6901                        if (start > offset)
6902                                ipf->ipf_hole_cnt++;
6903                        mp->b_cont = mp1->b_cont;
6904                        mp1->b_cont = mp;
6905                        mp1 = mp->b_cont;
6906                        offset = IP_REASS_START(mp1);
6907                        if (end >= offset) {
6908                                ipf->ipf_hole_cnt--;
6909                                /* Check for overlap. */
6910                                while (end > offset) {
6911                                        if (end < IP_REASS_END(mp1)) {
6912                                                mp->b_wptr -= end - offset;
6913                                                IP_REASS_SET_END(mp, offset);
6914                                                /*
6915                                                 * TODO we might bump
6916                                                 * this up twice if there is
6917                                                 * overlap at both ends.
6918                                                 */
6919                                                BUMP_MIB(ill->ill_ip_mib,
6920                                                    ipIfStatsReasmPartDups);
6921                                                break;
6922                                        }
6923                                        /* Did we cover another hole? */
6924                                        if ((mp1->b_cont &&
6925                                            IP_REASS_END(mp1)
```



```
6926                                            != IP_REASS_START(mp1->b_cont) &&
6927                                            end >=
6928                                            IP_REASS_START(mp1->b_cont)) ||
6929                                            (!ipf->ipf_last_frag_seen &&
6930                                            !more)) {
6931                                                ipf->ipf_hole_cnt--;
6932                                        }
6933                                        /* Clip out mp1. */
6934                                        if ((mp->b_cont = mp1->b_cont) ==
6935                                            NULL) {
6936                                                /*
6937                                                 * After clipping out mp1,
6938                                                 * this guy is now hanging
6939                                                 * off the end.
6940                                                 */
6941                                                ipf->ipf_tail_mp = mp;
6942                                        }
6943                                        IP_REASS_SET_START(mp1, 0);
6944                                        IP_REASS_SET_END(mp1, 0);
6945                                        /* Subtract byte count */
6946                                        ipf->ipf_count -=
6947                                            mp1->b_datap->db_lim -
6948                                            mp1->b_datap->db_base;
6949                                        freeb(mp1);
6950                                        BUMP_MIB(ill->ill_ip_mib,
6951                                            ipIfStatsReasmPartDups);
6952                                        mp1 = mp->b_cont;
6953                                        if (!mp1)
6954                                                break;
6955                                        offset = IP_REASS_START(mp1);
6956                                }
6957                        }
6958                        break;
6959                }
6960        } while (start = end, mp = next_mp);

6962        /* Fragment just processed could be the last one. Remember this fact */
6963        if (!more)
6964                ipf->ipf_last_frag_seen = B_TRUE;

6966        /* Still got holes? */
6967        if (ipf->ipf_hole_cnt)
6968                return (IP_REASS_PARTIAL);
6969        /* Clean up overloaded fields to avoid upstream disasters. */
6970        for (mp1 = ipf->ipf_mp->b_cont; mp1; mp1 = mp1->b_cont) {
6971                IP_REASS_SET_START(mp1, 0);
6972                IP_REASS_SET_END(mp1, 0);
6973        }
6974        return (IP_REASS_COMPLETE);
6975 }

6977 /*
6978  * Fragmentation reassembly.  Each ILL has a hash table for
6979  * queuing packets undergoing reassembly for all IPIFs
6980  * associated with the ILL.  The hash is based on the packet
6981  * IP ident field.  The ILL frag hash table was allocated
6982  * as a timer block at the time the ILL was created.  Whenever
6983  * there is anything on the reassembly queue, the timer will
6984  * be running.  Returns the reassembled packet if reassembly completes.
6985  */
6986 mblk_t *
6987 ip_input_fragment(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
6988 {
6989        uint32_t        frag_offset_flags;
6990        mblk_t          *t_mp;
6991        ipaddr_t        dst;
```

```
6992        uint8_t         proto = ipha->ipha_protocol;
6993        uint32_t        sum_val;
6994        uint16_t        sum_flags;
6995        ipf_t           *ipf;
6996        ipf_t           **ipfp;
6997        ipfb_t          *ipfb;
6998        uint16_t        ident;
6999        uint32_t        offset;
7000        ipaddr_t        src;
7001        uint_t          hdr_length;
7002        uint32_t        end;
7003        mblk_t          *mp1;
7004        mblk_t          *tail_mp;
7005        size_t          count;
7006        size_t          msg_len;
7007        uint8_t         ecn_info = 0;
7008        uint32_t        packet_size;
7009        boolean_t       pruned = B_FALSE;
7010        ill_t           *ill = ira->ira_ill;
7011        ip_stack_t      *ipst = ill->ill_ipst;

7013        /*
7014         * Drop the fragmented as early as possible, if
7015         * we don't have resource(s) to re-assemble.
7016         */
7017        if (ipst->ips_ip_reass_queue_bytes == 0) {
7018                freemsg(mp);
7019                return (NULL);
7020        }

7022        /* Check for fragmentation offset; return if there's none */
7023        if ((frag_offset_flags = ntohs(ipha->ipha_fragment_offset_and_flags) &
7024            (IPH_MF | IPH_OFFSET)) == 0)
7025                return (mp);

7027        /*
7028         * We utilize hardware computed checksum info only for UDP since
7029         * IP fragmentation is a normal occurrence for the protocol.  In
7030         * addition, checksum offload support for IP fragments carrying
7031         * UDP payload is commonly implemented across network adapters.
7032         */
7033        ASSERT(ira->ira_rill != NULL);
7034        if (proto == IPPROTO_UDP && dohwcksum &&
7035            ILL_HCKSUM_CAPABLE(ira->ira_rill) &&
7036            (DB_CKSUMFLAGS(mp) & (HCK_FULLCKSUM | HCK_PARTIALCKSUM))) {
7037                mblk_t *mp1 = mp->b_cont;
7038                int32_t len;

7040                /* Record checksum information from the packet */
7041                sum_val = (uint32_t)DB_CKSUM16(mp);
7042                sum_flags = DB_CKSUMFLAGS(mp);

7044                /* IP payload offset from beginning of mblk */
7045                offset = ((uchar_t *)ipha + IPH_HDR_LENGTH(ipha)) - mp->b_rptr;

7047                if ((sum_flags & HCK_PARTIALCKSUM) &&
7048                    (mp1 == NULL || mp1->b_cont == NULL) &&
7049                    offset >= DB_CKSUMSTART(mp) &&
7050                    ((len = offset - DB_CKSUMSTART(mp)) & 1) == 0) {
7051                        uint32_t adj;
7052                        /*
7053                         * Partial checksum has been calculated by hardware
7054                         * and attached to the packet; in addition, any
7055                         * prepended extraneous data is even byte aligned.
7056                         * If any such data exists, we adjust the checksum;
7057                         * this would also handle any postpended data.
```



```
7058                         */
7059                        IP_ADJCKSUM_PARTIAL(mp->b_rptr + DB_CKSUMSTART(mp),
7060                            mp, mp1, len, adj);

7062                        /* One's complement subtract extraneous checksum */
7063                        if (adj >= sum_val)
7064                                sum_val = ~(adj - sum_val) & 0xFFFF;
7065                        else
7066                                sum_val -= adj;
7067                }
7068        } else {
7069                sum_val = 0;
7070                sum_flags = 0;
7071        }

7073        /* Clear hardware checksumming flag */
7074        DB_CKSUMFLAGS(mp) = 0;

7076        ident = ipha->ipha_ident;
7077        offset = (frag_offset_flags << 3) & 0xFFFF;
7078        src = ipha->ipha_src;
7079        dst = ipha->ipha_dst;
7080        hdr_length = IPH_HDR_LENGTH(ipha);
7081        end = ntohs(ipha->ipha_length) - hdr_length;

7083        /* If end == 0 then we have a packet with no data, so just free it */
7084        if (end == 0) {
7085                freemsg(mp);
7086                return (NULL);
7087        }

7089        /* Record the ECN field info. */
7090        ecn_info = (ipha->ipha_type_of_service & 0x3);
7091        if (offset != 0) {
7092                /*
7093                 * If this isn't the first piece, strip the header, and
7094                 * add the offset to the end value.
7095                 */
7096                mp->b_rptr += hdr_length;
7097                end += offset;
7098        }

7100        /* Handle vnic loopback of fragments */
7101        if (mp->b_datap->db_ref > 2)
7102                msg_len = 0;
7103        else
7104                msg_len = MBLKSIZE(mp);

7106        tail_mp = mp;
7107        while (tail_mp->b_cont != NULL) {
7108                tail_mp = tail_mp->b_cont;
7109                if (tail_mp->b_datap->db_ref <= 2)
7110                        msg_len += MBLKSIZE(tail_mp);
7111        }

7113        /* If the reassembly list for this ILL will get too big, prune it */
7114        if ((msg_len + sizeof (*ipf) + ill->ill_frag_count) >=
7115            ipst->ips_ip_reass_queue_bytes) {
7116                DTRACE_PROBE3(ip_reass_queue_bytes, uint_t, msg_len,
7117                    uint_t, ill->ill_frag_count,
7118                    uint_t, ipst->ips_ip_reass_queue_bytes);
7119                ill_frag_prune(ill,
7120                    (ipst->ips_ip_reass_queue_bytes < msg_len) ? 0 :
7121                    (ipst->ips_ip_reass_queue_bytes - msg_len));
7122                pruned = B_TRUE;
7123        }
```

```
7125         ipfb = &ill->ill_frag_hash_tbl[ILL_FRAG_HASH(src, ident)];
7126         mutex_enter(&ipfb->ipfb_lock);

7128         ipfp = &ipfb->ipfb_ipf;
7129         /* Try to find an existing fragment queue for this packet. */
7130         for (;;) {
7131                 ipf = ipfp[0];
7132                 if (ipf != NULL) {
7133                         /*
7134                          * It has to match on ident and src/dst address.
7135                          */
7136                         if (ipf->ipf_ident == ident &&
7137                             ipf->ipf_src == src &&
7138                             ipf->ipf_dst == dst &&
7139                             ipf->ipf_protocol == proto) {
7140                                 /*
7141                                  * If we have received too many
7142                                  * duplicate fragments for this packet
7143                                  * free it.
7144                                  */
7145                                 if (ipf->ipf_num_dups > ip_max_frag_dups) {
7146                                         ill_frag_free_pkts(ill, ipfb, ipf, 1);
7147                                         freemsg(mp);
7148                                         mutex_exit(&ipfb->ipfb_lock);
7149                                         return (NULL);
7150                                 }
7151                                 /* Found it. */
7152                                 break;
7153                         }
7154                         ipfp = &ipf->ipf_hash_next;
7155                         continue;
7156                 }

7158                 /*
7159                  * If we pruned the list, do we want to store this new
7160                  * fragment?. We apply an optimization here based on the
7161                  * fact that most fragments will be received in order.
7162                  * So if the offset of this incoming fragment is zero,
7163                  * it is the first fragment of a new packet. We will
7164                  * keep it.  Otherwise drop the fragment, as we have
7165                  * probably pruned the packet already (since the
7166                  * packet cannot be found).
7167                  */
7168                 if (pruned && offset != 0) {
7169                         mutex_exit(&ipfb->ipfb_lock);
7170                         freemsg(mp);
7171                         return (NULL);
7172                 }

7174                 if (ipfb->ipfb_frag_pkts >= MAX_FRAG_PKTS(ipst))  {
7175                         /*
7176                          * Too many fragmented packets in this hash
7177                          * bucket. Free the oldest.
7178                          */
7179                         ill_frag_free_pkts(ill, ipfb, ipfb->ipfb_ipf, 1);
7180                 }

7182                 /* New guy.  Allocate a frag message. */
7183                 mp1 = allocb(sizeof (*ipf), BPRI_MED);
7184                 if (mp1 == NULL) {
7185                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7186                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
7187                         freemsg(mp);
7188 reass_done:
7189                         mutex_exit(&ipfb->ipfb_lock);
```



```
7190                         return (NULL);
7191                 }

7193                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsReasmReqds);
7194                 mp1->b_cont = mp;

7196                 /* Initialize the fragment header. */
7197                 ipf = (ipf_t *)mp1->b_rptr;
7198                 ipf->ipf_mp = mp1;
7199                 ipf->ipf_ptphn = ipfp;
7200                 ipfp[0] = ipf;
7201                 ipf->ipf_hash_next = NULL;
7202                 ipf->ipf_ident = ident;
7203                 ipf->ipf_protocol = proto;
7204                 ipf->ipf_src = src;
7205                 ipf->ipf_dst = dst;
7206                 ipf->ipf_nf_hdr_len = 0;
7207                 /* Record reassembly start time. */
7208                 ipf->ipf_timestamp = gethrestime_sec();
7209                 /* Record ipf generation and account for frag header */
7210                 ipf->ipf_gen = ill->ill_ipf_gen++;
7211                 ipf->ipf_count = MBLKSIZE(mp1);
7212                 ipf->ipf_last_frag_seen = B_FALSE;
7213                 ipf->ipf_ecn = ecn_info;
7214                 ipf->ipf_num_dups = 0;
7215                 ipfb->ipfb_frag_pkts++;
7216                 ipf->ipf_checksum = 0;
7217                 ipf->ipf_checksum_flags = 0;

7219                 /* Store checksum value in fragment header */
7220                 if (sum_flags != 0) {
7221                         sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7222                         sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7223                         ipf->ipf_checksum = sum_val;
7224                         ipf->ipf_checksum_flags = sum_flags;
7225                 }

7227                 /*
7228                  * We handle reassembly two ways.  In the easy case,
7229                  * where all the fragments show up in order, we do
7230                  * minimal bookkeeping, and just clip new pieces on
7231                  * the end.  If we ever see a hole, then we go off
7232                  * to ip_reassemble which has to mark the pieces and
7233                  * keep track of the number of holes, etc.  Obviously,
7234                  * the point of having both mechanisms is so we can
7235                  * handle the easy case as efficiently as possible.
7236                  */
7237                 if (offset == 0) {
7238                         /* Easy case, in-order reassembly so far. */
7239                         ipf->ipf_count += msg_len;
7240                         ipf->ipf_tail_mp = tail_mp;
7241                         /*
7242                          * Keep track of next expected offset in
7243                          * ipf_end.
7244                          */
7245                         ipf->ipf_end = end;
7246                         ipf->ipf_nf_hdr_len = hdr_length;
7247                 } else {
7248                         /* Hard case, hole at the beginning. */
7249                         ipf->ipf_tail_mp = NULL;
7250                         /*
7251                          * ipf_end == 0 means that we have given up
7252                          * on easy reassembly.
7253                          */
7254                         ipf->ipf_end = 0;
```

```
7256                        /* Forget checksum offload from now on */
7257                        ipf->ipf_checksum_flags = 0;

7259                        /*
7260                         * ipf_hole_cnt is set by ip_reassemble.
7261                         * ipf_count is updated by ip_reassemble.
7262                         * No need to check for return value here
7263                         * as we don't expect reassembly to complete
7264                         * or fail for the first fragment itself.
7265                         */
7266                        (void) ip_reassemble(mp, ipf,
7267                            (frag_offset_flags & IPH_OFFSET) << 3,
7268                            (frag_offset_flags & IPH_MF), ill, msg_len);
7269                }
7270                /* Update per ipfb and ill byte counts */
7271                ipfb->ipfb_count += ipf->ipf_count;
7272                ASSERT(ipfb->ipfb_count > 0);   /* Wraparound */
7273                atomic_add_32(&ill->ill_frag_count, ipf->ipf_count);
7274                /* If the frag timer wasn't already going, start it. */
7275                mutex_enter(&ill->ill_lock);
7276                ill_frag_timer_start(ill);
7277                mutex_exit(&ill->ill_lock);
7278                goto reass_done;
7279        }

7281        /*
7282         * If the packet's flag has changed (it could be coming up
7283         * from an interface different than the previous, therefore
7284         * possibly different checksum capability), then forget about
7285         * any stored checksum states.  Otherwise add the value to
7286         * the existing one stored in the fragment header.
7287         */
7288        if (sum_flags != 0 && sum_flags == ipf->ipf_checksum_flags) {
7289                sum_val += ipf->ipf_checksum;
7290                sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7291                sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7292                ipf->ipf_checksum = sum_val;
7293        } else if (ipf->ipf_checksum_flags != 0) {
7294                /* Forget checksum offload from now on */
7295                ipf->ipf_checksum_flags = 0;
7296        }

7298        /*
7299         * We have a new piece of a datagram which is already being
7300         * reassembled.  Update the ECN info if all IP fragments
7301         * are ECN capable.  If there is one which is not, clear
7302         * all the info.  If there is at least one which has CE
7303         * code point, IP needs to report that up to transport.
7304         */
7305        if (ecn_info != IPH_ECN_NECT && ipf->ipf_ecn != IPH_ECN_NECT) {
7306                if (ecn_info == IPH_ECN_CE)
7307                        ipf->ipf_ecn = IPH_ECN_CE;
7308        } else {
7309                ipf->ipf_ecn = IPH_ECN_NECT;
7310        }
7311        if (offset && ipf->ipf_end == offset) {
7312                /* The new fragment fits at the end */
7313                ipf->ipf_tail_mp->b_cont = mp;
7314                /* Update the byte count */
7315                ipf->ipf_count += msg_len;
7316                /* Update per ipfb and ill byte counts */
7317                ipfb->ipfb_count += msg_len;
7318                ASSERT(ipfb->ipfb_count > 0);   /* Wraparound */
7319                atomic_add_32(&ill->ill_frag_count, msg_len);
7320                if (frag_offset_flags & IPH_MF) {
7321                        /* More to come. */
```



```
7322                        ipf->ipf_end = end;
7323                        ipf->ipf_tail_mp = tail_mp;
7324                        goto reass_done;
7325                }
7326        } else {
7327                /* Go do the hard cases. */
7328                int ret;

7330                if (offset == 0)
7331                        ipf->ipf_nf_hdr_len = hdr_length;

7333                /* Save current byte count */
7334                count = ipf->ipf_count;
7335                ret = ip_reassemble(mp, ipf,
7336                    (frag_offset_flags & IPH_OFFSET) << 3,
7337                    (frag_offset_flags & IPH_MF), ill, msg_len);
7338                /* Count of bytes added and subtracted (freeb()ed) */
7339                count = ipf->ipf_count - count;
7340                if (count) {
7341                        /* Update per ipfb and ill byte counts */
7342                        ipfb->ipfb_count += count;
7343                        ASSERT(ipfb->ipfb_count > 0); /* Wraparound */
7344                        atomic_add_32(&ill->ill_frag_count, count);
7345                }
7346                if (ret == IP_REASS_PARTIAL) {
7347                        goto reass_done;
7348                } else if (ret == IP_REASS_FAILED) {
7349                        /* Reassembly failed. Free up all resources */
7350                        ill_frag_free_pkts(ill, ipfb, ipf, 1);
7351                        for (t_mp = mp; t_mp != NULL; t_mp = t_mp->b_cont) {
7352                                IP_REASS_SET_START(t_mp, 0);
7353                                IP_REASS_SET_END(t_mp, 0);
7354                        }
7355                        freemsg(mp);
7356                        goto reass_done;
7357                }
7358                /* We will reach here iff 'ret' is IP_REASS_COMPLETE */
7359        }
7360        /*
7361         * We have completed reassembly.  Unhook the frag header from
7362         * the reassembly list.
7363         *
7364         * Before we free the frag header, record the ECN info
7365         * to report back to the transport.
7366         */
7367        ecn_info = ipf->ipf_ecn;
7368        BUMP_MIB(ill->ill_ip_mib, ipIfStatsReasmOKs);
7369        ipfp = ipf->ipf_ptphn;

7371        /* We need to supply these to caller */
7372        if ((sum_flags = ipf->ipf_checksum_flags) != 0)
7373                sum_val = ipf->ipf_checksum;
7374        else
7375                sum_val = 0;

7377        mp1 = ipf->ipf_mp;
7378        count = ipf->ipf_count;
7379        ipf = ipf->ipf_hash_next;
7380        if (ipf != NULL)
7381                ipf->ipf_ptphn = ipfp;
7382        ipfp[0] = ipf;
7383        atomic_add_32(&ill->ill_frag_count, -count);
7384        ASSERT(ipfb->ipfb_count >= count);
7385        ipfb->ipfb_count -= count;
7386        ipfb->ipfb_frag_pkts--;
7387        mutex_exit(&ipfb->ipfb_lock);
```

```
7388         /* Ditch the frag header. */
7389         mp = mp1->b_cont;

7391         freeb(mp1);

7393         /* Restore original IP length in header. */
7394         packet_size = (uint32_t)msgdsize(mp);
7395         if (packet_size > IP_MAXPACKET) {
7396                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7397                 ip_drop_input("Reassembled packet too large", mp, ill);
7398                 freemsg(mp);
7399                 return (NULL);
7400         }

7402         if (DB_REF(mp) > 1) {
7403                 mblk_t *mp2 = copymsg(mp);

7405                 if (mp2 == NULL) {
7406                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7407                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
7408                         freemsg(mp);
7409                         return (NULL);
7410                 }
7411                 freemsg(mp);
7412                 mp = mp2;
7413         }
7414         ipha = (ipha_t *)mp->b_rptr;

7416         ipha->ipha_length = htons((uint16_t)packet_size);
7417         /* We're now complete, zip the frag state */
7418         ipha->ipha_fragment_offset_and_flags = 0;
7419         /* Record the ECN info. */
7420         ipha->ipha_type_of_service &= 0xFC;
7421         ipha->ipha_type_of_service |= ecn_info;

7423         /* Update the receive attributes */
7424         ira->ira_pktlen = packet_size;
7425         ira->ira_ip_hdr_length = IPH_HDR_LENGTH(ipha);

7427         /* Reassembly is successful; set checksum information in packet */
7428         DB_CKSUM16(mp) = (uint16_t)sum_val;
7429         DB_CKSUMFLAGS(mp) = sum_flags;
7430         DB_CKSUMSTART(mp) = ira->ira_ip_hdr_length;

7432         return (mp);
7433 }

7435 /*
7436  * Pullup function that should be used for IP input in order to
7437  * ensure we do not loose the L2 source address; we need the l2 source
7438  * address for IP_RECVSLLA and for ndp_input.
7439  *
7440  * We return either NULL or b_rptr.
7441  */
7442 void *
7443 ip_pullup(mblk_t *mp, ssize_t len, ip_recv_attr_t *ira)
7444 {
7445         ill_t           *ill = ira->ira_ill;

7447         if (ip_rput_pullups++ == 0) {
7448                 (void) mi_strlog(ill->ill_rq, 1, SL_ERROR|SL_TRACE,
7449                     "ip_pullup: %s forced us to "
7450                     " pullup pkt, hdr len %ld, hdr addr %p",
7451                     ill->ill_name, len, (void *)mp->b_rptr);
7452         }
7453         if (!(ira->ira_flags & IRAF_L2SRC_SET))
```



```
7454                 ip_setl2src(mp, ira, ira->ira_rill);
7455         ASSERT(ira->ira_flags & IRAF_L2SRC_SET);
7456         if (!pullupmsg(mp, len))
7457                 return (NULL);
7458         else
7459                 return (mp->b_rptr);
7460 }

7462 /*
7463  * Make sure ira_l2src has an address. If we don't have one fill with zeros.
7464  * When called from the ULP ira_rill will be NULL hence the caller has to
7465  * pass in the ill.
7466  */
7467 /* ARGSUSED */
7468 void
7469 ip_setl2src(mblk_t *mp, ip_recv_attr_t *ira, ill_t *ill)
7470 {
7471         const uchar_t *addr;
7472         int alen;

7474         if (ira->ira_flags & IRAF_L2SRC_SET)
7475                 return;

7477         ASSERT(ill != NULL);
7478         alen = ill->ill_phys_addr_length;
7479         ASSERT(alen <= sizeof (ira->ira_l2src));
7480         if (ira->ira_mhip != NULL &&
7481             (addr = ira->ira_mhip->mhi_saddr) != NULL) {
7482                 bcopy(addr, ira->ira_l2src, alen);
7483         } else if ((ira->ira_flags & IRAF_L2SRC_LOOPBACK) &&
7484             (addr = ill->ill_phys_addr) != NULL) {
7485                 bcopy(addr, ira->ira_l2src, alen);
7486         } else {
7487                 bzero(ira->ira_l2src, alen);
7488         }
7489         ira->ira_flags |= IRAF_L2SRC_SET;
7490 }

7492 /*
7493  * check ip header length and align it.
7494  */
7495 mblk_t *
7496 ip_check_and_align_header(mblk_t *mp, uint_t min_size, ip_recv_attr_t *ira)
7497 {
7498         ill_t   *ill = ira->ira_ill;
7499         ssize_t len;

7501         len = MBLKL(mp);

7503         if (!OK_32PTR(mp->b_rptr))
7504                 IP_STAT(ill->ill_ipst, ip_notaligned);
7505         else
7506                 IP_STAT(ill->ill_ipst, ip_recv_pullup);

7508         /* Guard against bogus device drivers */
7509         if (len < 0) {
7510                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7511                 ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7512                 freemsg(mp);
7513                 return (NULL);
7514         }

7516         if (len == 0) {
7517                 /* GLD sometimes sends up mblk with b_rptr == b_wptr! */
7518                 mblk_t *mp1 = mp->b_cont;
```

```
7520                 if (!(ira->ira_flags & IRAF_L2SRC_SET))
7521                         ip_setl2src(mp, ira, ira->ira_rill);
7522                 ASSERT(ira->ira_flags & IRAF_L2SRC_SET);

7524                 freeb(mp);
7525                 mp = mp1;
7526                 if (mp == NULL)
7527                         return (NULL);

7529                 if (OK_32PTR(mp->b_rptr) && MBLKL(mp) >= min_size)
7530                         return (mp);
7531         }
7532         if (ip_pullup(mp, min_size, ira) == NULL) {
7533                 if (msgdsize(mp) < min_size) {
7534                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7535                         ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7536                 } else {
7537                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7538                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
7539                 }
7540                 freemsg(mp);
7541                 return (NULL);
7542         }
7543         return (mp);
7544 }

7546 /*
7547  * Common code for IPv4 and IPv6 to check and pullup multi-mblks
7548  */
7549 mblk_t *
7550 ip_check_length(mblk_t *mp, uchar_t *rptr, ssize_t len, uint_t pkt_len,
7551     uint_t min_size, ip_recv_attr_t *ira)
7552 {
7553         ill_t   *ill = ira->ira_ill;

7555         /*
7556          * Make sure we have data length consistent
7557          * with the IP header.
7558          */
7559         if (mp->b_cont == NULL) {
7560                 /* pkt_len is based on ipha_len, not the mblk length */
7561                 if (pkt_len < min_size) {
7562                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7563                         ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7564                         freemsg(mp);
7565                         return (NULL);
7566                 }
7567                 if (len < 0) {
7568                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
7569                         ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
7570                         freemsg(mp);
7571                         return (NULL);
7572                 }
7573                 /* Drop any pad */
7574                 mp->b_wptr = rptr + pkt_len;
7575         } else if ((len += msgdsize(mp->b_cont)) != 0) {
7576                 ASSERT(pkt_len >= min_size);
7577                 if (pkt_len < min_size) {
7578                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7579                         ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7580                         freemsg(mp);
7581                         return (NULL);
7582                 }
7583                 if (len < 0) {
7584                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
7585                         ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
```



```
7586                         freemsg(mp);
7587                         return (NULL);
7588                 }
7589                 /* Drop any pad */
7590                 (void) adjmsg(mp, -len);
7591                 /*
7592                  * adjmsg may have freed an mblk from the chain, hence
7593                  * invalidate any hw checksum here. This will force IP to
7594                  * calculate the checksum in sw, but only for this packet.
7595                  */
7596                 DB_CKSUMFLAGS(mp) = 0;
7597                 IP_STAT(ill->ill_ipst, ip_multimblk);
7598         }
7599         return (mp);
7600 }

7602 /*
7603  * Check that the IPv4 opt_len is consistent with the packet and pullup
7604  * the options.
7605  */
7606 mblk_t *
7607 ip_check_optlen(mblk_t *mp, ipha_t *ipha, uint_t opt_len, uint_t pkt_len,
7608     ip_recv_attr_t *ira)
7609 {
7610         ill_t   *ill = ira->ira_ill;
7611         ssize_t len;

7613         /* Assume no IPv6 packets arrive over the IPv4 queue */
7614         if (IPH_HDR_VERSION(ipha) != IPV4_VERSION) {
7615                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7616                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInWrongIPVersion);
7617                 ip_drop_input("IPvN packet on IPv4 ill", mp, ill);
7618                 freemsg(mp);
7619                 return (NULL);
7620         }

7622         if (opt_len > (15 - IP_SIMPLE_HDR_LENGTH_IN_WORDS)) {
7623                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7624                 ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7625                 freemsg(mp);
7626                 return (NULL);
7627         }
7628         /*
7629          * Recompute complete header length and make sure we
7630          * have access to all of it.
7631          */
7632         len = ((size_t)opt_len + IP_SIMPLE_HDR_LENGTH_IN_WORDS) << 2;
7633         if (len > (mp->b_wptr - mp->b_rptr)) {
7634                 if (len > pkt_len) {
7635                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7636                         ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7637                         freemsg(mp);
7638                         return (NULL);
7639                 }
7640                 if (ip_pullup(mp, len, ira) == NULL) {
7641                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7642                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
7643                         freemsg(mp);
7644                         return (NULL);
7645                 }
7646         }
7647         return (mp);
7648 }

7650 /*
7651  * Returns a new ire, or the same ire, or NULL.
```

```
 * If a different IRE is returned, then it is held; the caller
 * needs to release it.
 * In no case is there any hold/release on the ire argument.
 */
ire_t *
ip_check_multihome(void *addr, ire_t *ire, ill_t *ill)
{
        ire_t           *new_ire;
        ill_t           *ire_ill;
        uint_t          ifindex;
        ip_stack_t      *ipst = ill->ill_ipst;
        boolean_t       strict_check = B_FALSE;

        /*
         * IPMP common case: if IRE and ILL are in the same group, there's no
         * issue (e.g. packet received on an underlying interface matched an
         * IRE_LOCAL on its associated group interface).
         */
        ASSERT(ire->ire_ill != NULL);
        if (IS_IN_SAME_ILLGRP(ill, ire->ire_ill))
                return (ire);

        /*
         * Do another ire lookup here, using the ingress ill, to see if the
         * interface is in a usesrc group.
         * As long as the ills belong to the same group, we don't consider
         * them to be arriving on the wrong interface. Thus, if the switch
         * is doing inbound load spreading, we won't drop packets when the
         * ip*_strict_dst_multihoming switch is on.
         * We also need to check for IPIF_UNNUMBERED point2point interfaces
         * where the local address may not be unique. In this case we were
         * at the mercy of the initial ire lookup and the IRE_LOCAL it
         * actually returned. The new lookup, which is more specific, should
         * only find the IRE_LOCAL associated with the ingress ill if one
         * exists.
         */
        if (ire->ire_ipversion == IPV4_VERSION) {
                if (ipst->ips_ip_strict_dst_multihoming)
                        strict_check = B_TRUE;
                new_ire = ire_ftable_lookup_v4(*((ipaddr_t *)addr), 0, 0,
                    IRE_LOCAL, ill, ALL_ZONES, NULL,
                    (MATCH_IRE_TYPE|MATCH_IRE_ILL), 0, ipst, NULL);
        } else {
                ASSERT(!IN6_IS_ADDR_MULTICAST((in6_addr_t *)addr));
                if (ipst->ips_ipv6_strict_dst_multihoming)
                        strict_check = B_TRUE;
                new_ire = ire_ftable_lookup_v6((in6_addr_t *)addr, NULL, NULL,
                    IRE_LOCAL, ill, ALL_ZONES, NULL,
                    (MATCH_IRE_TYPE|MATCH_IRE_ILL), 0, ipst, NULL);
        }
        /*
         * If the same ire that was returned in ip_input() is found then this
         * is an indication that usesrc groups are in use. The packet
         * arrived on a different ill in the group than the one associated with
         * the destination address.  If a different ire was found then the same
         * IP address must be hosted on multiple ills. This is possible with
         * unnumbered point2point interfaces. We switch to use this new ire in
         * order to have accurate interface statistics.
         */
        if (new_ire != NULL) {
                /* Note: held in one case but not the other? Caller handles */
                if (new_ire != ire)
                        return (new_ire);
                /* Unchanged */
                ire_refrele(new_ire);
                return (ire);
```



```
        }

        /*
         * Chase pointers once and store locally.
         */
        ASSERT(ire->ire_ill != NULL);
        ire_ill = ire->ire_ill;
        ifindex = ill->ill_usesrc_ifindex;

        /*
         * Check if it's a legal address on the 'usesrc' interface.
         * For IPMP data addresses the IRE_LOCAL is the upper, hence we
         * can just check phyint_ifindex.
         */
        if (ifindex != 0 && ifindex == ire_ill->ill_phyint->phyint_ifindex) {
                return (ire);
        }

        /*
         * If the ip*_strict_dst_multihoming switch is on then we can
         * only accept this packet if the interface is marked as routing.
         */
        if (!(strict_check))
                return (ire);

        if ((ill->ill_flags & ire->ire_ill->ill_flags & ILLF_ROUTER) != 0) {
                return (ire);
        }
        return (NULL);
}

/*
 * This function is used to construct a mac_header_info_s from a
 * DL_UNITDATA_IND message.
 * The address fields in the mhi structure points into the message,
 * thus the caller can't use those fields after freeing the message.
 *
 * We determine whether the packet received is a non-unicast packet
 * and in doing so, determine whether or not it is broadcast vs multicast.
 * For it to be a broadcast packet, we must have the appropriate mblk_t
 * hanging off the ill_t.  If this is either not present or doesn't match
 * the destination mac address in the DL_UNITDATA_IND, the packet is deemed
 * to be multicast.  Thus NICs that have no broadcast address (or no
 * capability for one, such as point to point links) cannot return as
 * the packet being broadcast.
 */
void
ip_dlur_to_mhi(ill_t *ill, mblk_t *mb, struct mac_header_info_s *mhip)
{
        dl_unitdata_ind_t *ind = (dl_unitdata_ind_t *)mb->b_rptr;
        mblk_t *bmp;
        uint_t extra_offset;

        bzero(mhip, sizeof (struct mac_header_info_s));

        mhip->mhi_dsttype = MAC_ADDRTYPE_UNICAST;

        if (ill->ill_sap_length < 0)
                extra_offset = 0;
        else
                extra_offset = ill->ill_sap_length;

        mhip->mhi_daddr = (uchar_t *)ind + ind->dl_dest_addr_offset +
            extra_offset;
        mhip->mhi_saddr = (uchar_t *)ind + ind->dl_src_addr_offset +
            extra_offset;
```

```
7785         if (!ind->dl_group_address)
7786                 return;

7788         /* Multicast or broadcast */
7789         mhip->mhi_dsttype = MAC_ADDRTYPE_MULTICAST;

7791         if (ind->dl_dest_addr_offset > sizeof (*ind) &&
7792             ind->dl_dest_addr_offset + ind->dl_dest_addr_length < MBLKL(mb) &&
7793             (bmp = ill->ill_bcast_mp) != NULL) {
7794                 dl_unitdata_req_t *dlur;
7795                 uint8_t *bphys_addr;

7797                 dlur = (dl_unitdata_req_t *)bmp->b_rptr;
7798                 bphys_addr = (uchar_t *)dlur + dlur->dl_dest_addr_offset +
7799                     extra_offset;

7801                 if (bcmp(mhip->mhi_daddr, bphys_addr,
7802                     ind->dl_dest_addr_length) == 0)
7803                         mhip->mhi_dsttype = MAC_ADDRTYPE_BROADCAST;
7804         }
7805 }

7807 /*
7808  * This function is used to construct a mac_header_info_s from a
7809  * M_DATA fastpath message from a DLPI driver.
7810  * The address fields in the mhi structure points into the message,
7811  * thus the caller can't use those fields after freeing the message.
7812  *
7813  * We determine whether the packet received is a non-unicast packet
7814  * and in doing so, determine whether or not it is broadcast vs multicast.
7815  * For it to be a broadcast packet, we must have the appropriate mblk_t
7816  * hanging off the ill_t.  If this is either not present or doesn't match
7817  * the destination mac address in the DL_UNITDATA_IND, the packet is deemed
7818  * to be multicast.  Thus NICs that have no broadcast address (or no
7819  * capability for one, such as point to point links) cannot return as
7820  * the packet being broadcast.
7821  */
7822 void
7823 ip_mdata_to_mhi(ill_t *ill, mblk_t *mp, struct mac_header_info_s *mhip)
7824 {
7825         mblk_t *bmp;
7826         struct ether_header *pether;

7828         bzero(mhip, sizeof (struct mac_header_info_s));

7830         mhip->mhi_dsttype = MAC_ADDRTYPE_UNICAST;

7832         pether = (struct ether_header *)((char *)mp->b_rptr
7833             - sizeof (struct ether_header));

7835         /*
7836          * Make sure the interface is an ethernet type, since we don't
7837          * know the header format for anything but Ethernet. Also make
7838          * sure we are pointing correctly above db_base.
7839          */
7840         if (ill->ill_type != IFT_ETHER)
7841                 return;

7843 retry:
7844         if ((uchar_t *)pether < mp->b_datap->db_base)
7845                 return;

7847         /* Is there a VLAN tag? */
7848         if (ill->ill_isv6) {
7849                 if (pether->ether_type != htons(ETHERTYPE_IPV6)) {
```



```
7850                         pether = (struct ether_header *)((char *)pether - 4);
7851                         goto retry;
7852                 }
7853         } else {
7854                 if (pether->ether_type != htons(ETHERTYPE_IP)) {
7855                         pether = (struct ether_header *)((char *)pether - 4);
7856                         goto retry;
7857                 }
7858         }
7859         mhip->mhi_daddr = (uchar_t *)&pether->ether_dhost;
7860         mhip->mhi_saddr = (uchar_t *)&pether->ether_shost;

7862         if (!(mhip->mhi_daddr[0] & 0x01))
7863                 return;

7865         /* Multicast or broadcast */
7866         mhip->mhi_dsttype = MAC_ADDRTYPE_MULTICAST;

7868         if ((bmp = ill->ill_bcast_mp) != NULL) {
7869                 dl_unitdata_req_t *dlur;
7870                 uint8_t *bphys_addr;
7871                 uint_t  addrlen;

7873                 dlur = (dl_unitdata_req_t *)bmp->b_rptr;
7874                 addrlen = dlur->dl_dest_addr_length;
7875                 if (ill->ill_sap_length < 0) {
7876                         bphys_addr = (uchar_t *)dlur +
7877                             dlur->dl_dest_addr_offset;
7878                         addrlen += ill->ill_sap_length;
7879                 } else {
7880                         bphys_addr = (uchar_t *)dlur +
7881                             dlur->dl_dest_addr_offset +
7882                             ill->ill_sap_length;
7883                         addrlen -= ill->ill_sap_length;
7884                 }
7885                 if (bcmp(mhip->mhi_daddr, bphys_addr, addrlen) == 0)
7886                         mhip->mhi_dsttype = MAC_ADDRTYPE_BROADCAST;
7887         }
7888 }

7890 /*
7891  * Handle anything but M_DATA messages
7892  * We see the DL_UNITDATA_IND which are part
7893  * of the data path, and also the other messages from the driver.
7894  */
7895 void
7896 ip_rput_notdata(ill_t *ill, mblk_t *mp)
7897 {
7898         mblk_t          *first_mp;
7899         struct iocblk   *iocp;
7900         struct mac_header_info_s mhi;

7902         switch (DB_TYPE(mp)) {
7903         case M_PROTO:
7904         case M_PCPROTO: {
7905                 if (((dl_unitdata_ind_t *)mp->b_rptr)->dl_primitive !=
7906                     DL_UNITDATA_IND) {
7907                         /* Go handle anything other than data elsewhere. */
7908                         ip_rput_dlpi(ill, mp);
7909                         return;
7910                 }

7912                 first_mp = mp;
7913                 mp = first_mp->b_cont;
7914                 first_mp->b_cont = NULL;
```

```
7916                 if (mp == NULL) {
7917                         freeb(first_mp);
7918                         return;
7919                 }
7920                 ip_dlur_to_mhi(ill, first_mp, &mhi);
7921                 if (ill->ill_isv6)
7922                         ip_input_v6(ill, NULL, mp, &mhi);
7923                 else
7924                         ip_input(ill, NULL, mp, &mhi);

7926                 /* Ditch the DLPI header. */
7927                 freeb(first_mp);
7928                 return;
7929         }
7930         case M_IOCACK:
7931                 iocp = (struct iocblk *)mp->b_rptr;
7932                 switch (iocp->ioc_cmd) {
7933                 case DL_IOC_HDR_INFO:
7934                         ill_fastpath_ack(ill, mp);
7935                         return;
7936                 default:
7937                         putnext(ill->ill_rq, mp);
7938                         return;
7939                 }
7940                 /* FALLTHRU */
7941         case M_ERROR:
7942         case M_HANGUP:
7943                 mutex_enter(&ill->ill_lock);
7944                 if (ill->ill_state_flags & ILL_CONDEMNED) {
7945                         mutex_exit(&ill->ill_lock);
7946                         freemsg(mp);
7947                         return;
7948                 }
7949                 ill_refhold_locked(ill);
7950                 mutex_exit(&ill->ill_lock);
7951                 qwriter_ip(ill, ill->ill_rq, mp, ip_rput_other, CUR_OP,
7952                     B_FALSE);
7953                 return;
7954         case M_CTL:
7955                 putnext(ill->ill_rq, mp);
7956                 return;
7957         case M_IOCNAK:
7958                 ip1dbg(("got iocnak "));
7959                 iocp = (struct iocblk *)mp->b_rptr;
7960                 switch (iocp->ioc_cmd) {
7961                 case DL_IOC_HDR_INFO:
7962                         ip_rput_other(NULL, ill->ill_rq, mp, NULL);
7963                         return;
7964                 default:
7965                         break;
7966                 }
7967                 /* FALLTHRU */
7968         default:
7969                 putnext(ill->ill_rq, mp);
7970                 return;
7971         }
7972 }

7974 /* Read side put procedure.  Packets coming from the wire arrive here. */
7975 void
7976 ip_rput(queue_t *q, mblk_t *mp)
7977 {
7978         ill_t   *ill;
7979         union DL_primitives *dl;

7981         ill = (ill_t *)q->q_ptr;
```



```
7983         if (ill->ill_state_flags & (ILL_CONDEMNED | ILL_LL_SUBNET_PENDING)) {
7984                 /*
7985                  * If things are opening or closing, only accept high-priority
7986                  * DLPI messages.  (On open ill->ill_ipif has not yet been
7987                  * created; on close, things hanging off the ill may have been
7988                  * freed already.)
7989                  */
7990                 dl = (union DL_primitives *)mp->b_rptr;
7991                 if (DB_TYPE(mp) != M_PCPROTO ||
7992                     dl->dl_primitive == DL_UNITDATA_IND) {
7993                         inet_freemsg(mp);
7994                         return;
7995                 }
7996         }
7997         if (DB_TYPE(mp) == M_DATA) {
7998                 struct mac_header_info_s mhi;

8000                 ip_mdata_to_mhi(ill, mp, &mhi);
8001                 ip_input(ill, NULL, mp, &mhi);
8002         } else {
8003                 ip_rput_notdata(ill, mp);
8004         }
8005 }

8007 /*
8008  * Move the information to a copy.
8009  */
8010 mblk_t *
8011 ip_fix_dbref(mblk_t *mp, ip_recv_attr_t *ira)
8012 {
8013         mblk_t          *mp1;
8014         ill_t           *ill = ira->ira_ill;
8015         ip_stack_t      *ipst = ill->ill_ipst;

8017         IP_STAT(ipst, ip_db_ref);

8019         /* Make sure we have ira_l2src before we loose the original mblk */
8020         if (!(ira->ira_flags & IRAF_L2SRC_SET))
8021                 ip_setl2src(mp, ira, ira->ira_rill);

8023         mp1 = copymsg(mp);
8024         if (mp1 == NULL) {
8025                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
8026                 ip_drop_input("ipIfStatsInDiscards", mp, ill);
8027                 freemsg(mp);
8028                 return (NULL);
8029         }
8030         /* preserve the hardware checksum flags and data, if present */
8031         if (DB_CKSUMFLAGS(mp) != 0) {
8032                 DB_CKSUMFLAGS(mp1) = DB_CKSUMFLAGS(mp);
8033                 DB_CKSUMSTART(mp1) = DB_CKSUMSTART(mp);
8034                 DB_CKSUMSTUFF(mp1) = DB_CKSUMSTUFF(mp);
8035                 DB_CKSUMEND(mp1) = DB_CKSUMEND(mp);
8036                 DB_CKSUM16(mp1) = DB_CKSUM16(mp);
8037         }
8038         freemsg(mp);
8039         return (mp1);
8040 }

8042 static void
8043 ip_dlpi_error(ill_t *ill, t_uscalar_t prim, t_uscalar_t dl_err,
8044     t_uscalar_t err)
8045 {
8046         if (dl_err == DL_SYSERR) {
8047                 (void) mi_strlog(ill->ill_rq, 1, SL_CONSOLE|SL_ERROR|SL_TRACE,
```

```
8048                    "%s: %s failed: DL_SYSERR (errno %u)\n",
8049                    ill->ill_name, dl_primstr(prim), err);
8050                return;
8051        }

8053        (void) mi_strlog(ill->ill_rq, 1, SL_CONSOLE|SL_ERROR|SL_TRACE,
8054            "%s: %s failed: %s\n", ill->ill_name, dl_primstr(prim),
8055            dl_errstr(dl_err));
8056 }

8058 /*
8059  * ip_rput_dlpi is called by ip_rput to handle all DLPI messages other
8060  * than DL_UNITDATA_IND messages. If we need to process this message
8061  * exclusively, we call qwriter_ip, in which case we also need to call
8062  * ill_refhold before that, since qwriter_ip does an ill_refrele.
8063  */
8064 void
8065 ip_rput_dlpi(ill_t *ill, mblk_t *mp)
8066 {
8067        dl_ok_ack_t     *dloa = (dl_ok_ack_t *)mp->b_rptr;
8068        dl_error_ack_t  *dlea = (dl_error_ack_t *)dloa;
8069        queue_t         *q = ill->ill_rq;
8070        t_uscalar_t     prim = dloa->dl_primitive;
8071        t_uscalar_t     reqprim = DL_PRIM_INVAL;

8073        DTRACE_PROBE3(ill__dlpi, char *, "ip_rput_dlpi",
8074            char *, dl_primstr(prim), ill_t *, ill);
8075        ip1dbg(("ip_rput_dlpi"));

8077        /*
8078         * If we received an ACK but didn't send a request for it, then it
8079         * can't be part of any pending operation; discard up-front.
8080         */
8081        switch (prim) {
8082        case DL_ERROR_ACK:
8083                reqprim = dlea->dl_error_primitive;
8084                ip2dbg(("ip_rput_dlpi(%s): DL_ERROR_ACK for %s (0x%x): %s "
8085                    "(0x%x), unix %u\n", ill->ill_name, dl_primstr(reqprim),
8086                    reqprim, dl_errstr(dlea->dl_errno), dlea->dl_errno,
8087                    dlea->dl_unix_errno));
8088                break;
8089        case DL_OK_ACK:
8090                reqprim = dloa->dl_correct_primitive;
8091                break;
8092        case DL_INFO_ACK:
8093                reqprim = DL_INFO_REQ;
8094                break;
8095        case DL_BIND_ACK:
8096                reqprim = DL_BIND_REQ;
8097                break;
8098        case DL_PHYS_ADDR_ACK:
8099                reqprim = DL_PHYS_ADDR_REQ;
8100                break;
8101        case DL_NOTIFY_ACK:
8102                reqprim = DL_NOTIFY_REQ;
8103                break;
8104        case DL_CAPABILITY_ACK:
8105                reqprim = DL_CAPABILITY_REQ;
8106                break;
8107        }

8109        if (prim != DL_NOTIFY_IND) {
8110                if (reqprim == DL_PRIM_INVAL ||
8111                    !ill_dlpi_pending(ill, reqprim)) {
8112                        /* Not a DLPI message we support or expected */
8113                        freemsg(mp);
```



```
8114                        return;
8115                }
8116                ip1dbg(("ip_rput: received %s for %s\n", dl_primstr(prim),
8117                    dl_primstr(reqprim)));
8118        }

8120        switch (reqprim) {
8121        case DL_UNBIND_REQ:
8122                /*
8123                 * NOTE: we mark the unbind as complete even if we got a
8124                 * DL_ERROR_ACK, since there's not much else we can do.
8125                 */
8126                mutex_enter(&ill->ill_lock);
8127                ill->ill_state_flags &= ~ILL_DL_UNBIND_IN_PROGRESS;
8128                cv_signal(&ill->ill_cv);
8129                mutex_exit(&ill->ill_lock);
8130                break;

8132        case DL_ENABMULTI_REQ:
8133                if (prim == DL_OK_ACK) {
8134                        if (ill->ill_dlpi_multicast_state == IDS_INPROGRESS)
8135                                ill->ill_dlpi_multicast_state = IDS_OK;
8136                }
8137                break;
8138        }

8140        /*
8141         * The message is one we're waiting for (or DL_NOTIFY_IND), but we
8142         * need to become writer to continue to process it.  Because an
8143         * exclusive operation doesn't complete until replies to all queued
8144         * DLPI messages have been received, we know we're in the middle of an
8145         * exclusive operation and pass CUR_OP (except for DL_NOTIFY_IND).
8146         *
8147         * As required by qwriter_ip(), we refhold the ill; it will refrele.
8148         * Since this is on the ill stream we unconditionally bump up the
8149         * refcount without doing ILL_CAN_LOOKUP().
8150         */
8151        ill_refhold(ill);
8152        if (prim == DL_NOTIFY_IND)
8153                qwriter_ip(ill, q, mp, ip_rput_dlpi_writer, NEW_OP, B_FALSE);
8154        else
8155                qwriter_ip(ill, q, mp, ip_rput_dlpi_writer, CUR_OP, B_FALSE);
8156 }

8158 /*
8159  * Handling of DLPI messages that require exclusive access to the ipsq.
8160  *
8161  * Need to do ipsq_pending_mp_get on ioctl completion, which could
8162  * happen here. (along with mi_copy_done)
8163  */
8164 /* ARGSUSED */
8165 static void
8166 ip_rput_dlpi_writer(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
8167 {
8168        dl_ok_ack_t     *dloa = (dl_ok_ack_t *)mp->b_rptr;
8169        dl_error_ack_t  *dlea = (dl_error_ack_t *)dloa;
8170        int             err = 0;
8171        ill_t           *ill = (ill_t *)q->q_ptr;
8172        ipif_t          *ipif = NULL;
8173        mblk_t          *mp1 = NULL;
8174        conn_t          *connp = NULL;
8175        t_uscalar_t     paddrreq;
8176        mblk_t          *mp_hw;
8177        boolean_t       success;
8178        boolean_t       ioctl_aborted = B_FALSE;
8179        boolean_t       log = B_TRUE;
```

```
8181         DTRACE_PROBE3(ill__dlpi, char *, "ip_rput_dlpi_writer",
8182             char *, dl_primstr(dloa->dl_primitive), ill_t *, ill);
8183
8184         ip1dbg(("ip_rput_dlpi_writer .."));
8185         ASSERT(ipsq->ipsq_xop == ill->ill_phyint->phyint_ipsq->ipsq_xop);
8186         ASSERT(IAM_WRITER_ILL(ill));
8187
8188         ipif = ipsq->ipsq_xop->ipx_pending_ipif;
8189         /*
8190          * The current ioctl could have been aborted by the user and a new
8191          * ioctl to bring up another ill could have started. We could still
8192          * get a response from the driver later.
8193          */
8194         if (ipif != NULL && ipif->ipif_ill != ill)
8195                 ioctl_aborted = B_TRUE;
8196
8197         switch (dloa->dl_primitive) {
8198         case DL_ERROR_ACK:
8199                 ip1dbg(("ip_rput_dlpi_writer: got DL_ERROR_ACK for %s\n",
8200                     dl_primstr(dlea->dl_error_primitive)));
8201
8202                 DTRACE_PROBE3(ill__dlpi, char *, "ip_rput_dlpi_writer error",
8203                     char *, dl_primstr(dlea->dl_error_primitive),
8204                     ill_t *, ill);
8205
8206                 switch (dlea->dl_error_primitive) {
8207                 case DL_DISABMULTI_REQ:
8208                         ill_dlpi_done(ill, dlea->dl_error_primitive);
8209                         break;
8210                 case DL_PROMISCON_REQ:
8211                 case DL_PROMISCOFF_REQ:
8212                 case DL_UNBIND_REQ:
8213                 case DL_ATTACH_REQ:
8214                 case DL_INFO_REQ:
8215                         ill_dlpi_done(ill, dlea->dl_error_primitive);
8216                         break;
8217                 case DL_NOTIFY_REQ:
8218                         ill_dlpi_done(ill, DL_NOTIFY_REQ);
8219                         log = B_FALSE;
8220                         break;
8221                 case DL_PHYS_ADDR_REQ:
8222                         /*
8223                          * For IPv6 only, there are two additional
8224                          * phys_addr_req's sent to the driver to get the
8225                          * IPv6 token and lla. This allows IP to acquire
8226                          * the hardware address format for a given interface
8227                          * without having built in knowledge of the hardware
8228                          * address. ill_phys_addr_pend keeps track of the last
8229                          * DL_PAR sent so we know which response we are
8230                          * dealing with. ill_dlpi_done will update
8231                          * ill_phys_addr_pend when it sends the next req.
8232                          * We don't complete the IOCTL until all three DL_PARs
8233                          * have been attempted, so set *_len to 0 and break.
8234                          */
8235                         paddrreq = ill->ill_phys_addr_pend;
8236                         ill_dlpi_done(ill, DL_PHYS_ADDR_REQ);
8237                         if (paddrreq == DL_IPV6_TOKEN) {
8238                                 ill->ill_token_length = 0;
8239                                 log = B_FALSE;
8240                                 break;
8241                         } else if (paddrreq == DL_IPV6_LINK_LAYER_ADDR) {
8242                                 ill->ill_nd_lla_len = 0;
8243                                 log = B_FALSE;
8244                                 break;
8245                         }
```



```
8246                         /*
8247                          * Something went wrong with the DL_PHYS_ADDR_REQ.
8248                          * We presumably have an IOCTL hanging out waiting
8249                          * for completion. Find it and complete the IOCTL
8250                          * with the error noted.
8251                          * However, ill_dl_phys was called on an ill queue
8252                          * (from SIOCSLIFNAME), thus conn_pending_ill is not
8253                          * set. But the ioctl is known to be pending on ill_wq.
8254                          */
8255                         if (!ill->ill_ifname_pending)
8256                                 break;
8257                         ill->ill_ifname_pending = 0;
8258                         if (!ioctl_aborted)
8259                                 mp1 = ipsq_pending_mp_get(ipsq, &connp);
8260                         if (mp1 != NULL) {
8261                                 /*
8262                                  * This operation (SIOCSLIFNAME) must have
8263                                  * happened on the ill. Assert there is no conn
8264                                  */
8265                                 ASSERT(connp == NULL);
8266                                 q = ill->ill_wq;
8267                         }
8268                         break;
8269                 case DL_BIND_REQ:
8270                         ill_dlpi_done(ill, DL_BIND_REQ);
8271                         if (ill->ill_ifname_pending)
8272                                 break;
8273                         mutex_enter(&ill->ill_lock);
8274                         ill->ill_state_flags &= ~ILL_DOWN_IN_PROGRESS;
8275                         mutex_exit(&ill->ill_lock);
8276                         /*
8277                          * Something went wrong with the bind.  We presumably
8278                          * have an IOCTL hanging out waiting for completion.
8279                          * Find it, take down the interface that was coming
8280                          * up, and complete the IOCTL with the error noted.
8281                          */
8282                         if (!ioctl_aborted)
8283                                 mp1 = ipsq_pending_mp_get(ipsq, &connp);
8284                         if (mp1 != NULL) {
8285                                 /*
8286                                  * This might be a result of a DL_NOTE_REPLUMB
8287                                  * notification. In that case, connp is NULL.
8288                                  */
8289                                 if (connp != NULL)
8290                                         q = CONNP_TO_WQ(connp);
8291
8292                                 (void) ipif_down(ipif, NULL, NULL);
8293                                 /* error is set below the switch */
8294                         }
8295                         break;
8296                 case DL_ENABMULTI_REQ:
8297                         ill_dlpi_done(ill, DL_ENABMULTI_REQ);
8298
8299                         if (ill->ill_dlpi_multicast_state == IDS_INPROGRESS)
8300                                 ill->ill_dlpi_multicast_state = IDS_FAILED;
8301                         if (ill->ill_dlpi_multicast_state == IDS_FAILED) {
8302
8303                                 printf("ip: joining multicasts failed (%d)"
8304                                     " on %s - will use link layer "
8305                                     "broadcasts for multicast\n",
8306                                     dlea->dl_errno, ill->ill_name);
8307
8308                                 /*
8309                                  * Set up for multi_bcast; We are the
8310                                  * writer, so ok to access ill->ill_ipif
8311                                  * without any lock.
```

```
8312                                 */
8313                                mutex_enter(&ill->ill_phyint->phyint_lock);
8314                                ill->ill_phyint->phyint_flags |=
8315                                    PHYI_MULTI_BCAST;
8316                                mutex_exit(&ill->ill_phyint->phyint_lock);

8318                        }
8319                        freemsg(mp);    /* Don't want to pass this up */
8320                        return;
8321                case DL_CAPABILITY_REQ:
8322                        ip1dbg(("ip_rput_dlpi_writer: got DL_ERROR_ACK for "
8323                            "DL_CAPABILITY REQ\n"));
8324                        if (ill->ill_dlpi_capab_state == IDCS_PROBE_SENT)
8325                                ill->ill_dlpi_capab_state = IDCS_FAILED;
8326                        ill_capability_done(ill);
8327                        freemsg(mp);
8328                        return;
8329                }
8330                /*
8331                 * Note the error for IOCTL completion (mp1 is set when
8332                 * ready to complete ioctl). If ill_ifname_pending_err is
8333                 * set, an error occured during plumbing (ill_ifname_pending),
8334                 * so we want to report that error.
8335                 *
8336                 * NOTE: there are two addtional DL_PHYS_ADDR_REQ's
8337                 * (DL_IPV6_TOKEN and DL_IPV6_LINK_LAYER_ADDR) that are
8338                 * expected to get errack'd if the driver doesn't support
8339                 * these flags (e.g. ethernet). log will be set to B_FALSE
8340                 * if these error conditions are encountered.
8341                 */
8342                if (mp1 != NULL) {
8343                        if (ill->ill_ifname_pending_err != 0)  {
8344                                err = ill->ill_ifname_pending_err;
8345                                ill->ill_ifname_pending_err = 0;
8346                        } else {
8347                                err = dlea->dl_unix_errno ?
8348                                    dlea->dl_unix_errno : ENXIO;
8349                        }
8350                /*
8351                 * If we're plumbing an interface and an error hasn't already
8352                 * been saved, set ill_ifname_pending_err to the error passed
8353                 * up. Ignore the error if log is B_FALSE (see comment above).
8354                 */
8355                } else if (log && ill->ill_ifname_pending &&
8356                    ill->ill_ifname_pending_err == 0) {
8357                        ill->ill_ifname_pending_err = dlea->dl_unix_errno ?
8358                            dlea->dl_unix_errno : ENXIO;
8359                }

8361                if (log)
8362                        ip_dlpi_error(ill, dlea->dl_error_primitive,
8363                            dlea->dl_errno, dlea->dl_unix_errno);
8364                break;
8365        case DL_CAPABILITY_ACK:
8366                ill_capability_ack(ill, mp);
8367                /*
8368                 * The message has been handed off to ill_capability_ack
8369                 * and must not be freed below
8370                 */
8371                mp = NULL;
8372                break;

8374        case DL_INFO_ACK:
8375                /* Call a routine to handle this one. */
8376                ill_dlpi_done(ill, DL_INFO_REQ);
8377                ip_ll_subnet_defaults(ill, mp);
```



```
8378                ASSERT(!MUTEX_HELD(&ill->ill_phyint->phyint_ipsq->ipsq_lock));
8379                return;
8380        case DL_BIND_ACK:
8381                /*
8382                 * We should have an IOCTL waiting on this unless
8383                 * sent by ill_dl_phys, in which case just return
8384                 */
8385                ill_dlpi_done(ill, DL_BIND_REQ);

8387                if (ill->ill_ifname_pending) {
8388                        DTRACE_PROBE2(ip__rput__dlpi__ifname__pending,
8389                            ill_t *, ill, mblk_t *, mp);
8390                        break;
8391                }
8392                mutex_enter(&ill->ill_lock);
8393                ill->ill_dl_up = 1;
8394                ill->ill_state_flags &= ~ILL_DOWN_IN_PROGRESS;
8395                mutex_exit(&ill->ill_lock);

8397                if (!ioctl_aborted)
8398                        mp1 = ipsq_pending_mp_get(ipsq, &connp);
8399                if (mp1 == NULL) {
8400                        DTRACE_PROBE1(ip__rput__dlpi__no__mblk, ill_t *, ill);
8401                        break;
8402                }
8403                /*
8404                 * mp1 was added by ill_dl_up(). if that is a result of
8405                 * a DL_NOTE_REPLUMB notification, connp could be NULL.
8406                 */
8407                if (connp != NULL)
8408                        q = CONNP_TO_WQ(connp);
8409                /*
8410                 * We are exclusive. So nothing can change even after
8411                 * we get the pending mp.
8412                 */
8413                ip1dbg(("ip_rput_dlpi: bind_ack %s\n", ill->ill_name));
8414                DTRACE_PROBE1(ip__rput__dlpi__bind__ack, ill_t *, ill);
8415                ill_nic_event_dispatch(ill, 0, NE_UP, NULL, 0);

8417                /*
8418                 * Now bring up the resolver; when that is complete, we'll
8419                 * create IREs.  Note that we intentionally mirror what
8420                 * ipif_up() would have done, because we got here by way of
8421                 * ill_dl_up(), which stopped ipif_up()'s processing.
8422                 */
8423                if (ill->ill_isv6) {
8424                        /*
8425                         * v6 interfaces.
8426                         * Unlike ARP which has to do another bind
8427                         * and attach, once we get here we are
8428                         * done with NDP
8429                         */
8430                        (void) ipif_resolver_up(ipif, Res_act_initial);
8431                        if ((err = ipif_ndp_up(ipif, B_TRUE)) == 0)
8432                                err = ipif_up_done_v6(ipif);
8433                } else if (ill->ill_net_type == IRE_IF_RESOLVER) {
8434                        /*
8435                         * ARP and other v4 external resolvers.
8436                         * Leave the pending mblk intact so that
8437                         * the ioctl completes in ip_rput().
8438                         */
8439                        if (connp != NULL)
8440                                mutex_enter(&connp->conn_lock);
8441                        mutex_enter(&ill->ill_lock);
8442                        success = ipsq_pending_mp_add(connp, ipif, q, mp1, 0);
8443                        mutex_exit(&ill->ill_lock);
```

```
8444                        if (connp != NULL)
8445                                mutex_exit(&connp->conn_lock);
8446                        if (success) {
8447                                err = ipif_resolver_up(ipif, Res_act_initial);
8448                                if (err == EINPROGRESS) {
8449                                        freemsg(mp);
8450                                        return;
8451                                }
8452                                mp1 = ipsq_pending_mp_get(ipsq, &connp);
8453                        } else {
8454                                /* The conn has started closing */
8455                                err = EINTR;
8456                        }
8457                } else {
8458                        /*
8459                         * This one is complete. Reply to pending ioctl.
8460                         */
8461                        (void) ipif_resolver_up(ipif, Res_act_initial);
8462                        err = ipif_up_done(ipif);
8463                }

8465                if ((err == 0) && (ill->ill_up_ipifs)) {
8466                        err = ill_up_ipifs(ill, q, mp1);
8467                        if (err == EINPROGRESS) {
8468                                freemsg(mp);
8469                                return;
8470                        }
8471                }

8473                /*
8474                 * If we have a moved ipif to bring up, and everything has
8475                 * succeeded to this point, bring it up on the IPMP ill.
8476                 * Otherwise, leave it down -- the admin can try to bring it
8477                 * up by hand if need be.
8478                 */
8479                if (ill->ill_move_ipif != NULL) {
8480                        if (err != 0) {
8481                                ill->ill_move_ipif = NULL;
8482                        } else {
8483                                ipif = ill->ill_move_ipif;
8484                                ill->ill_move_ipif = NULL;
8485                                err = ipif_up(ipif, q, mp1);
8486                                if (err == EINPROGRESS) {
8487                                        freemsg(mp);
8488                                        return;
8489                                }
8490                        }
8491                }
8492                break;

8494        case DL_NOTIFY_IND: {
8495                dl_notify_ind_t *notify = (dl_notify_ind_t *)mp->b_rptr;
8496                uint_t orig_mtu, orig_mc_mtu;

8498                switch (notify->dl_notification) {
8499                case DL_NOTE_PHYS_ADDR:
8500                        err = ill_set_phys_addr(ill, mp);
8501                        break;

8503                case DL_NOTE_REPLUMB:
8504                        /*
8505                         * Directly return after calling ill_replumb().
8506                         * Note that we should not free mp as it is reused
8507                         * in the ill_replumb() function.
8508                         */
8509                        err = ill_replumb(ill, mp);
```



```
8510                        return;

8512                case DL_NOTE_FASTPATH_FLUSH:
8513                        nce_flush(ill, B_FALSE);
8514                        break;

8516                case DL_NOTE_SDU_SIZE:
8517                case DL_NOTE_SDU_SIZE2:
8518                        /*
8519                         * The dce and fragmentation code can cope with
8520                         * this changing while packets are being sent.
8521                         * When packets are sent ip_output will discover
8522                         * a change.
8523                         *
8524                         * Change the MTU size of the interface.
8525                         */
8526                        mutex_enter(&ill->ill_lock);
8527                        orig_mtu = ill->ill_mtu;
8528                        orig_mc_mtu = ill->ill_mc_mtu;
8529                        switch (notify->dl_notification) {
8530                        case DL_NOTE_SDU_SIZE:
8531                                ill->ill_current_frag =
8532                                    (uint_t)notify->dl_data;
8533                                ill->ill_mc_mtu = (uint_t)notify->dl_data;
8534                                break;
8535                        case DL_NOTE_SDU_SIZE2:
8536                                ill->ill_current_frag =
8537                                    (uint_t)notify->dl_data1;
8538                                ill->ill_mc_mtu = (uint_t)notify->dl_data2;
8539                                break;
8540                        }
8541                        if (ill->ill_current_frag > ill->ill_max_frag)
8542                                ill->ill_max_frag = ill->ill_current_frag;

8544                        if (!(ill->ill_flags & ILLF_FIXEDMTU)) {
8545                                ill->ill_mtu = ill->ill_current_frag;

8547                                /*
8548                                 * If ill_user_mtu was set (via
8549                                 * SIOCSLIFLNKINFO), clamp ill_mtu at it.
8550                                 */
8551                                if (ill->ill_user_mtu != 0 &&
8552                                    ill->ill_user_mtu < ill->ill_mtu)
8553                                        ill->ill_mtu = ill->ill_user_mtu;

8555                                if (ill->ill_user_mtu != 0 &&
8556                                    ill->ill_user_mtu < ill->ill_mc_mtu)
8557                                        ill->ill_mc_mtu = ill->ill_user_mtu;

8559                                if (ill->ill_isv6) {
8560                                        if (ill->ill_mtu < IPV6_MIN_MTU)
8561                                                ill->ill_mtu = IPV6_MIN_MTU;
8562                                        if (ill->ill_mc_mtu < IPV6_MIN_MTU)
8563                                                ill->ill_mc_mtu = IPV6_MIN_MTU;
8564                                } else {
8565                                        if (ill->ill_mtu < IP_MIN_MTU)
8566                                                ill->ill_mtu = IP_MIN_MTU;
8567                                        if (ill->ill_mc_mtu < IP_MIN_MTU)
8568                                                ill->ill_mc_mtu = IP_MIN_MTU;
8569                                }
8570                        } else if (ill->ill_mc_mtu > ill->ill_mtu) {
8571                                ill->ill_mc_mtu = ill->ill_mtu;
8572                        }

8574                        mutex_exit(&ill->ill_lock);
8575                        /*
```

```
8576                         * Make sure all dce_generation checks find out
8577                         * that ill_mtu/ill_mc_mtu has changed.
8578                         */
8579                        if (orig_mtu != ill->ill_mtu ||
8580                            orig_mc_mtu != ill->ill_mc_mtu) {
8581                                dce_increment_all_generations(ill->ill_isv6,
8582                                    ill->ill_ipst);
8583                        }
8584
8585                        /*
8586                         * Refresh IPMP meta-interface MTU if necessary.
8587                         */
8588                        if (IS_UNDER_IPMP(ill))
8589                                ipmp_illgrp_refresh_mtu(ill->ill_grp);
8590                        break;
8591
8592                case DL_NOTE_LINK_UP:
8593                case DL_NOTE_LINK_DOWN: {
8594                        /*
8595                         * We are writer. ill / phyint / ipsq assocs stable.
8596                         * The RUNNING flag reflects the state of the link.
8597                         */
8598                        phyint_t *phyint = ill->ill_phyint;
8599                        uint64_t new_phyint_flags;
8600                        boolean_t changed = B_FALSE;
8601                        boolean_t went_up;
8602
8603                        went_up = notify->dl_notification == DL_NOTE_LINK_UP;
8604                        mutex_enter(&phyint->phyint_lock);
8605
8606                        new_phyint_flags = went_up ?
8607                            phyint->phyint_flags | PHYI_RUNNING :
8608                            phyint->phyint_flags & ~PHYI_RUNNING;
8609
8610                        if (IS_IPMP(ill)) {
8611                                new_phyint_flags = went_up ?
8612                                    new_phyint_flags & ~PHYI_FAILED :
8613                                    new_phyint_flags | PHYI_FAILED;
8614                        }
8615
8616                        if (new_phyint_flags != phyint->phyint_flags) {
8617                                phyint->phyint_flags = new_phyint_flags;
8618                                changed = B_TRUE;
8619                        }
8620                        mutex_exit(&phyint->phyint_lock);
8621                        /*
8622                         * ill_restart_dad handles the DAD restart and routing
8623                         * socket notification logic.
8624                         */
8625                        if (changed) {
8626                                ill_restart_dad(phyint->phyint_illv4, went_up);
8627                                ill_restart_dad(phyint->phyint_illv6, went_up);
8628                        }
8629                        break;
8630                }
8631                case DL_NOTE_PROMISC_ON_PHYS: {
8632                        phyint_t *phyint = ill->ill_phyint;
8633
8634                        mutex_enter(&phyint->phyint_lock);
8635                        phyint->phyint_flags |= PHYI_PROMISC;
8636                        mutex_exit(&phyint->phyint_lock);
8637                        break;
8638                }
8639                case DL_NOTE_PROMISC_OFF_PHYS: {
8640                        phyint_t *phyint = ill->ill_phyint;
```



```
8642                        mutex_enter(&phyint->phyint_lock);
8643                        phyint->phyint_flags &= ~PHYI_PROMISC;
8644                        mutex_exit(&phyint->phyint_lock);
8645                        break;
8646                }
8647                case DL_NOTE_CAPAB_RENEG:
8648                        /*
8649                         * Something changed on the driver side.
8650                         * It wants us to renegotiate the capabilities
8651                         * on this ill. One possible cause is the aggregation
8652                         * interface under us where a port got added or
8653                         * went away.
8654                         *
8655                         * If the capability negotiation is already done
8656                         * or is in progress, reset the capabilities and
8657                         * mark the ill's ill_capab_reneg to be B_TRUE,
8658                         * so that when the ack comes back, we can start
8659                         * the renegotiation process.
8660                         *
8661                         * Note that if ill_capab_reneg is already B_TRUE
8662                         * (ill_dlpi_capab_state is IDS_UNKNOWN in this case),
8663                         * the capability resetting request has been sent
8664                         * and the renegotiation has not been started yet;
8665                         * nothing needs to be done in this case.
8666                         */
8667                        ipsq_current_start(ipsq, ill->ill_ipif, 0);
8668                        ill_capability_reset(ill, B_TRUE);
8669                        ipsq_current_finish(ipsq);
8670                        break;
8671
8672                case DL_NOTE_ALLOWED_IPS:
8673                        ill_set_allowed_ips(ill, mp);
8674                        break;
8675                default:
8676                        ip0dbg(("ip_rput_dlpi_writer: unknown notification "
8677                            "type 0x%x for DL_NOTIFY_IND\n",
8678                            notify->dl_notification));
8679                        break;
8680                }
8681
8682                /*
8683                 * As this is an asynchronous operation, we
8684                 * should not call ill_dlpi_done
8685                 */
8686                break;
8687        }
8688        case DL_NOTIFY_ACK: {
8689                dl_notify_ack_t *noteack = (dl_notify_ack_t *)mp->b_rptr;
8690
8691                if (noteack->dl_notifications & DL_NOTE_LINK_UP)
8692                        ill->ill_note_link = 1;
8693                ill_dlpi_done(ill, DL_NOTIFY_REQ);
8694                break;
8695        }
8696        case DL_PHYS_ADDR_ACK: {
8697                /*
8698                 * As part of plumbing the interface via SIOCSLIFNAME,
8699                 * ill_dl_phys() will queue a series of DL_PHYS_ADDR_REQs,
8700                 * whose answers we receive here.  As each answer is received,
8701                 * we call ill_dlpi_done() to dispatch the next request as
8702                 * we're processing the current one.  Once all answers have
8703                 * been received, we use ipsq_pending_mp_get() to dequeue the
8704                 * outstanding IOCTL and reply to it.  (Because ill_dl_phys()
8705                 * is invoked from an ill queue, conn_oper_pending_ill is not
8706                 * available, but we know the ioctl is pending on ill_wq.)
8707                 */
```

```
8708                uint_t  paddrlen, paddroff;
8709                uint8_t *addr;

8711                paddrreq = ill->ill_phys_addr_pend;
8712                paddrlen = ((dl_phys_addr_ack_t *)mp->b_rptr)->dl_addr_length;
8713                paddroff = ((dl_phys_addr_ack_t *)mp->b_rptr)->dl_addr_offset;
8714                addr = mp->b_rptr + paddroff;

8716                ill_dlpi_done(ill, DL_PHYS_ADDR_REQ);
8717                if (paddrreq == DL_IPV6_TOKEN) {
8718                        /*
8719                         * bcopy to low-order bits of ill_token
8720                         *
8721                         * XXX Temporary hack - currently, all known tokens
8722                         * are 64 bits, so I'll cheat for the moment.
8723                         */
8724                        bcopy(addr, &ill->ill_token.s6_addr32[2], paddrlen);
8725                        ill->ill_token_length = paddrlen;
8726                        break;
8727                } else if (paddrreq == DL_IPV6_LINK_LAYER_ADDR) {
8728                        ASSERT(ill->ill_nd_lla_mp == NULL);
8729                        ill_set_ndmp(ill, mp, paddroff, paddrlen);
8730                        mp = NULL;
8731                        break;
8732                } else if (paddrreq == DL_CURR_DEST_ADDR) {
8733                        ASSERT(ill->ill_dest_addr_mp == NULL);
8734                        ill->ill_dest_addr_mp = mp;
8735                        ill->ill_dest_addr = addr;
8736                        mp = NULL;
8737                        if (ill->ill_isv6) {
8738                                ill_setdesttoken(ill);
8739                                ipif_setdestlinklocal(ill->ill_ipif);
8740                        }
8741                        break;
8742                }

8744                ASSERT(paddrreq == DL_CURR_PHYS_ADDR);
8745                ASSERT(ill->ill_phys_addr_mp == NULL);
8746                if (!ill->ill_ifname_pending)
8747                        break;
8748                ill->ill_ifname_pending = 0;
8749                if (!ioctl_aborted)
8750                        mp1 = ipsq_pending_mp_get(ipsq, &connp);
8751                if (mp1 != NULL) {
8752                        ASSERT(connp == NULL);
8753                        q = ill->ill_wq;
8754                }
8755                /*
8756                 * If any error acks received during the plumbing sequence,
8757                 * ill_ifname_pending_err will be set. Break out and send up
8758                 * the error to the pending ioctl.
8759                 */
8760                if (ill->ill_ifname_pending_err != 0) {
8761                        err = ill->ill_ifname_pending_err;
8762                        ill->ill_ifname_pending_err = 0;
8763                        break;
8764                }

8766                ill->ill_phys_addr_mp = mp;
8767                ill->ill_phys_addr = (paddrlen == 0 ? NULL : addr);
8768                mp = NULL;

8770                /*
8771                 * If paddrlen or ill_phys_addr_length is zero, the DLPI
8772                 * provider doesn't support physical addresses.  We check both
8773                 * paddrlen and ill_phys_addr_length because sppp (PPP) does
```



```
8774                 * not have physical addresses, but historically adversises a
8775                 * physical address length of 0 in its DL_INFO_ACK, but 6 in
8776                 * its DL_PHYS_ADDR_ACK.
8777                 */
8778                if (paddrlen == 0 || ill->ill_phys_addr_length == 0) {
8779                        ill->ill_phys_addr = NULL;
8780                } else if (paddrlen != ill->ill_phys_addr_length) {
8781                        ip0dbg(("DL_PHYS_ADDR_ACK: got addrlen %d, expected %d",
8782                            paddrlen, ill->ill_phys_addr_length));
8783                        err = EINVAL;
8784                        break;
8785                }

8787                if (ill->ill_nd_lla_mp == NULL) {
8788                        if ((mp_hw = copyb(ill->ill_phys_addr_mp)) == NULL) {
8789                                err = ENOMEM;
8790                                break;
8791                        }
8792                        ill_set_ndmp(ill, mp_hw, paddroff, paddrlen);
8793                }

8795                if (ill->ill_isv6) {
8796                        ill_setdefaulttoken(ill);
8797                        ipif_setlinklocal(ill->ill_ipif);
8798                }
8799                break;
8800        }
8801        case DL_OK_ACK:
8802                ip2dbg(("DL_OK_ACK %s (0x%x)\n",
8803                    dl_primstr((int)dloa->dl_correct_primitive),
8804                    dloa->dl_correct_primitive));
8805                DTRACE_PROBE3(ill__dlpi, char *, "ip_rput_dlpi_writer ok",
8806                    char *, dl_primstr(dloa->dl_correct_primitive),
8807                    ill_t *, ill);

8809                switch (dloa->dl_correct_primitive) {
8810                case DL_ENABMULTI_REQ:
8811                case DL_DISABMULTI_REQ:
8812                        ill_dlpi_done(ill, dloa->dl_correct_primitive);
8813                        break;
8814                case DL_PROMISCON_REQ:
8815                case DL_PROMISCOFF_REQ:
8816                case DL_UNBIND_REQ:
8817                case DL_ATTACH_REQ:
8818                        ill_dlpi_done(ill, dloa->dl_correct_primitive);
8819                        break;
8820                }
8821                break;
8822        default:
8823                break;
8824        }

8826        freemsg(mp);
8827        if (mp1 == NULL)
8828                return;

8830        /*
8831         * The operation must complete without EINPROGRESS since
8832         * ipsq_pending_mp_get() has removed the mblk (mp1).  Otherwise,
8833         * the operation will be stuck forever inside the IPSQ.
8834         */
8835        ASSERT(err != EINPROGRESS);

8837        DTRACE_PROBE4(ipif__ioctl, char *, "ip_rput_dlpi_writer finish",
8838            int, ipsq->ipsq_xop->ipx_current_ioctl, ill_t *, ill,
8839            ipif_t *, NULL);
```

```
8841         switch (ipsq->ipsq_xop->ipx_current_ioctl) {
8842         case 0:
8843                 ipsq_current_finish(ipsq);
8844                 break;

8846         case SIOCSLIFNAME:
8847         case IF_UNITSEL: {
8848                 ill_t *ill_other = ILL_OTHER(ill);

8850                 /*
8851                  * If SIOCSLIFNAME or IF_UNITSEL is about to succeed, and the
8852                  * ill has a peer which is in an IPMP group, then place ill
8853                  * into the same group.  One catch: although ifconfig plumbs
8854                  * the appropriate IPMP meta-interface prior to plumbing this
8855                  * ill, it is possible for multiple ifconfig applications to
8856                  * race (or for another application to adjust plumbing), in
8857                  * which case the IPMP meta-interface we need will be missing.
8858                  * If so, kick the phyint out of the group.
8859                  */
8860                 if (err == 0 && ill_other != NULL && IS_UNDER_IPMP(ill_other)) {
8861                         ipmp_grp_t      *grp = ill->ill_phyint->phyint_grp;
8862                         ipmp_illgrp_t   *illg;

8864                         illg = ill->ill_isv6 ? grp->gr_v6 : grp->gr_v4;
8865                         if (illg == NULL)
8866                                 ipmp_phyint_leave_grp(ill->ill_phyint);
8867                         else
8868                                 ipmp_ill_join_illgrp(ill, illg);
8869                 }

8871                 if (ipsq->ipsq_xop->ipx_current_ioctl == IF_UNITSEL)
8872                         ip_ioctl_finish(q, mp1, err, NO_COPYOUT, ipsq);
8873                 else
8874                         ip_ioctl_finish(q, mp1, err, COPYOUT, ipsq);
8875                 break;
8876         }
8877         case SIOCLIFADDIF:
8878                 ip_ioctl_finish(q, mp1, err, COPYOUT, ipsq);
8879                 break;

8881         default:
8882                 ip_ioctl_finish(q, mp1, err, NO_COPYOUT, ipsq);
8883                 break;
8884         }
8885 }

8887 /*
8888  * ip_rput_other is called by ip_rput to handle messages modifying the global
8889  * state in IP.  If 'ipsq' is non-NULL, caller is writer on it.
8890  */
8891 /* ARGSUSED */
8892 void
8893 ip_rput_other(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
8894 {
8895         ill_t           *ill = q->q_ptr;
8896         struct iocblk   *iocp;

8898         ip1dbg(("ip_rput_other "));
8899         if (ipsq != NULL) {
8900                 ASSERT(IAM_WRITER_IPSQ(ipsq));
8901                 ASSERT(ipsq->ipsq_xop ==
8902                     ill->ill_phyint->phyint_ipsq->ipsq_xop);
8903         }

8905         switch (mp->b_datap->db_type) {
```



```
8906         case M_ERROR:
8907         case M_HANGUP:
8908                 /*
8909                  * The device has a problem.  We force the ILL down.  It can
8910                  * be brought up again manually using SIOCSIFFLAGS (via
8911                  * ifconfig or equivalent).
8912                  */
8913                 ASSERT(ipsq != NULL);
8914                 if (mp->b_rptr < mp->b_wptr)
8915                         ill->ill_error = (int)(*mp->b_rptr & 0xFF);
8916                 if (ill->ill_error == 0)
8917                         ill->ill_error = ENXIO;
8918                 if (!ill_down_start(q, mp))
8919                         return;
8920                 ipif_all_down_tail(ipsq, q, mp, NULL);
8921                 break;
8922         case M_IOCNAK: {
8923                 iocp = (struct iocblk *)mp->b_rptr;

8925                 ASSERT(iocp->ioc_cmd == DL_IOC_HDR_INFO);
8926                 /*
8927                  * If this was the first attempt, turn off the fastpath
8928                  * probing.
8929                  */
8930                 mutex_enter(&ill->ill_lock);
8931                 if (ill->ill_dlpi_fastpath_state == IDS_INPROGRESS) {
8932                         ill->ill_dlpi_fastpath_state = IDS_FAILED;
8933                         mutex_exit(&ill->ill_lock);
8934                         /*
8935                          * don't flush the nce_t entries: we use them
8936                          * as an index to the ncec itself.
8937                          */
8938                         ip1dbg(("ip_rput: DLPI fastpath off on interface %s\n",
8939                             ill->ill_name));
8940                 } else {
8941                         mutex_exit(&ill->ill_lock);
8942                 }
8943                 freemsg(mp);
8944                 break;
8945         }
8946         default:
8947                 ASSERT(0);
8948                 break;
8949         }
8950 }

8952 /*
8953  * Update any source route, record route or timestamp options
8954  * When it fails it has consumed the message and BUMPed the MIB.
8955  */
8956 boolean_t
8957 ip_forward_options(mblk_t *mp, ipha_t *ipha, ill_t *dst_ill,
8958     ip_recv_attr_t *ira)
8959 {
8960         ipoptp_t        opts;
8961         uchar_t         *opt;
8962         uint8_t         optval;
8963         uint8_t         optlen;
8964         ipaddr_t        dst;
8965         ipaddr_t        ifaddr;
8966         uint32_t        ts;
8967         timestruc_t     now;
8968         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;

8970         ip2dbg(("ip_forward_options\n"));
8971         dst = ipha->ipha_dst;
```

```
8972         for (optval = ipoptp_first(&opts, ipha);
8973             optval != IPOPT_EOL;
8974             optval = ipoptp_next(&opts)) {
8975                 ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
8976                 opt = opts.ipoptp_cur;
8977                 optlen = opts.ipoptp_len;
8978                 ip2dbg(("ip_forward_options: opt %d, len %d\n",
8979                     optval, opts.ipoptp_len));
8980                 switch (optval) {
8981                         uint32_t off;
8982                 case IPOPT_SSRR:
8983                 case IPOPT_LSRR:
8984                         /* Check if adminstratively disabled */
8985                         if (!ipst->ips_ip_forward_src_routed) {
8986                                 BUMP_MIB(dst_ill->ill_ip_mib,
8987                                     ipIfStatsForwProhibits);
8988                                 ip_drop_input("ICMP_SOURCE_ROUTE_FAILED",
8989                                     mp, dst_ill);
8990                                 icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED,
8991                                     ira);
8992                                 return (B_FALSE);
8993                         }
8994                         if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
8995                                 /*
8996                                  * Must be partial since ip_input_options
8997                                  * checked for strict.
8998                                  */
8999                                 break;
9000                         }
9001                         off = opt[IPOPT_OFFSET];
9002                         off--;
9003                 redo_srr:
9004                         if (optlen < IP_ADDR_LEN ||
9005                             off > optlen - IP_ADDR_LEN) {
9006                                 /* End of source route */
9007                                 ip1dbg((
9008                                     "ip_forward_options: end of SR\n"));
9009                                 break;
9010                         }
9011                         /* Pick a reasonable address on the outbound if */
9012                         ASSERT(dst_ill != NULL);
9013                         if (ip_select_source_v4(dst_ill, INADDR_ANY, dst,
9014                             INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9015                             NULL) != 0) {
9016                                 /* No source! Shouldn't happen */
9017                                 ifaddr = INADDR_ANY;
9018                         }
9019                         bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9020                         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9021                         ip1dbg(("ip_forward_options: next hop 0x%x\n",
9022                             ntohl(dst)));

9024                         /*
9025                          * Check if our address is present more than
9026                          * once as consecutive hops in source route.
9027                          */
9028                         if (ip_type_v4(dst, ipst) == IRE_LOCAL) {
9029                                 off += IP_ADDR_LEN;
9030                                 opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9031                                 goto redo_srr;
9032                         }
9033                         ipha->ipha_dst = dst;
9034                         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9035                         break;
9036                 case IPOPT_RR:
9037                         off = opt[IPOPT_OFFSET];
```



```
9038                         off--;
9039                         if (optlen < IP_ADDR_LEN ||
9040                             off > optlen - IP_ADDR_LEN) {
9041                                 /* No more room - ignore */
9042                                 ip1dbg((
9043                                     "ip_forward_options: end of RR\n"));
9044                                 break;
9045                         }
9046                         /* Pick a reasonable address on the outbound if */
9047                         ASSERT(dst_ill != NULL);
9048                         if (ip_select_source_v4(dst_ill, INADDR_ANY, dst,
9049                             INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9050                             NULL) != 0) {
9051                                 /* No source! Shouldn't happen */
9052                                 ifaddr = INADDR_ANY;
9053                         }
9054                         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9055                         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9056                         break;
9057                 case IPOPT_TS:
9058                         /* Insert timestamp if there is room */
9059                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9060                         case IPOPT_TS_TSONLY:
9061                                 off = IPOPT_TS_TIMELEN;
9062                                 break;
9063                         case IPOPT_TS_PRESPEC:
9064                         case IPOPT_TS_PRESPEC_RFC791:
9065                                 /* Verify that the address matched */
9066                                 off = opt[IPOPT_OFFSET] - 1;
9067                                 bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9068                                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9069                                         /* Not for us */
9070                                         break;
9071                                 }
9072                                 /* FALLTHRU */
9073                         case IPOPT_TS_TSANDADDR:
9074                                 off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
9075                                 break;
9076                         default:
9077                                 /*
9078                                  * ip_*put_options should have already
9079                                  * dropped this packet.
9080                                  */
9081                                 cmn_err(CE_PANIC, "ip_forward_options: "
9082                                     "unknown IT - bug in ip_input_options?\n");
9083                                 return (B_TRUE);        /* Keep "lint" happy */
9084                         }
9085                         if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
9086                                 /* Increase overflow counter */
9087                                 off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
9088                                 opt[IPOPT_POS_OV_FLG] =
9089                                     (uint8_t)((opt[IPOPT_POS_OV_FLG] & 0x0F) |
9090                                     (off << 4));
9091                                 break;
9092                         }
9093                         off = opt[IPOPT_OFFSET] - 1;
9094                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9095                         case IPOPT_TS_PRESPEC:
9096                         case IPOPT_TS_PRESPEC_RFC791:
9097                         case IPOPT_TS_TSANDADDR:
9098                                 /* Pick a reasonable addr on the outbound if */
9099                                 ASSERT(dst_ill != NULL);
9100                                 if (ip_select_source_v4(dst_ill, INADDR_ANY,
9101                                     dst, INADDR_ANY, ALL_ZONES, ipst, &ifaddr,
9102                                     NULL, NULL) != 0) {
9103                                         /* No source! Shouldn't happen */
```

```
9104                                    ifaddr = INADDR_ANY;
9105                            }
9106                            bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9107                            opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9108                            /* FALLTHRU */
9109                    case IPOPT_TS_TSONLY:
9110                            off = opt[IPOPT_OFFSET] - 1;
9111                            /* Compute # of milliseconds since midnight */
9112                            gethrestime(&now);
9113                            ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
9114                                now.tv_nsec / (NANOSEC / MILLISEC);
9115                            bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
9116                            opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
9117                            break;
9118                    }
9119                    break;
9120            }
9121    }
9122    return (B_TRUE);
9123 }

9125 /*
9126  * Call ill_frag_timeout to do garbage collection. ill_frag_timeout
9127  * returns 'true' if there are still fragments left on the queue, in
9128  * which case we restart the timer.
9129  */
9130 void
9131 ill_frag_timer(void *arg)
9132 {
9133    ill_t   *ill = (ill_t *)arg;
9134    boolean_t frag_pending;
9135    ip_stack_t *ipst = ill->ill_ipst;
9136    time_t  timeout;

9138    mutex_enter(&ill->ill_lock);
9139    ASSERT(!ill->ill_fragtimer_executing);
9140    if (ill->ill_state_flags & ILL_CONDEMNED) {
9141            ill->ill_frag_timer_id = 0;
9142            mutex_exit(&ill->ill_lock);
9143            return;
9144    }
9145    ill->ill_fragtimer_executing = 1;
9146    mutex_exit(&ill->ill_lock);

9148    timeout = (ill->ill_isv6 ? ipst->ips_ipv6_reassembly_timeout :
9149        ipst->ips_ip_reassembly_timeout);

9151    frag_pending = ill_frag_timeout(ill, timeout);

9153    /*
9154     * Restart the timer, if we have fragments pending or if someone
9155     * wanted us to be scheduled again.
9156     */
9157    mutex_enter(&ill->ill_lock);
9158    ill->ill_fragtimer_executing = 0;
9159    ill->ill_frag_timer_id = 0;
9160    if (frag_pending || ill->ill_fragtimer_needrestart)
9161            ill_frag_timer_start(ill);
9162    mutex_exit(&ill->ill_lock);
9163 }

9165 void
9166 ill_frag_timer_start(ill_t *ill)
9167 {
9168    ip_stack_t *ipst = ill->ill_ipst;
9169    clock_t timeo_ms;
```



```
9171    ASSERT(MUTEX_HELD(&ill->ill_lock));

9173    /* If the ill is closing or opening don't proceed */
9174    if (ill->ill_state_flags & ILL_CONDEMNED)
9175            return;

9177    if (ill->ill_fragtimer_executing) {
9178            /*
9179             * ill_frag_timer is currently executing. Just record the
9180             * the fact that we want the timer to be restarted.
9181             * ill_frag_timer will post a timeout before it returns,
9182             * ensuring it will be called again.
9183             */
9184            ill->ill_fragtimer_needrestart = 1;
9185            return;
9186    }

9188    if (ill->ill_frag_timer_id == 0) {
9189            timeo_ms = (ill->ill_isv6 ? ipst->ips_ipv6_reassembly_timeout :
9190                ipst->ips_ip_reassembly_timeout) * SECONDS;

9192            /*
9193             * The timer is neither running nor is the timeout handler
9194             * executing. Post a timeout so that ill_frag_timer will be
9195             * called
9196             */
9197            ill->ill_frag_timer_id = timeout(ill_frag_timer, ill,
9198                MSEC_TO_TICK(timeo_ms >> 1));
9199            ill->ill_fragtimer_needrestart = 0;
9200    }
9201 }

9203 /*
9204  * Update any source route, record route or timestamp options.
9205  * Check that we are at end of strict source route.
9206  * The options have already been checked for sanity in ip_input_options().
9207  */
9208 boolean_t
9209 ip_input_local_options(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
9210 {
9211    ipoptp_t        opts;
9212    uchar_t         *opt;
9213    uint8_t         optval;
9214    uint8_t         optlen;
9215    ipaddr_t        dst;
9216    ipaddr_t        ifaddr;
9217    uint32_t        ts;
9218    timestruc_t     now;
9219    ill_t           *ill = ira->ira_ill;
9220    ip_stack_t      *ipst = ill->ill_ipst;

9222    ip2dbg(("ip_input_local_options\n"));

9224    for (optval = ipoptp_first(&opts, ipha);
9225        optval != IPOPT_EOL;
9226        optval = ipoptp_next(&opts)) {
9227            ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
9228            opt = opts.ipoptp_cur;
9229            optlen = opts.ipoptp_len;
9230            ip2dbg(("ip_input_local_options: opt %d, len %d\n",
9231                optval, optlen));
9232            switch (optval) {
9233                    uint32_t off;
9234            case IPOPT_SSRR:
9235            case IPOPT_LSRR:
```

```
9236                        off = opt[IPOPT_OFFSET];
9237                        off--;
9238                        if (optlen < IP_ADDR_LEN ||
9239                            off > optlen - IP_ADDR_LEN) {
9240                                /* End of source route */
9241                                ip1dbg(("ip_input_local_options: end of SR\n"));
9242                                break;
9243                        }
9244                        /*
9245                         * This will only happen if two consecutive entries
9246                         * in the source route contains our address or if
9247                         * it is a packet with a loose source route which
9248                         * reaches us before consuming the whole source route
9249                         */
9250                        ip1dbg(("ip_input_local_options: not end of SR\n"));
9251                        if (optval == IPOPT_SSRR) {
9252                                goto bad_src_route;
9253                        }
9254                        /*
9255                         * Hack: instead of dropping the packet truncate the
9256                         * source route to what has been used by filling the
9257                         * rest with IPOPT_NOP.
9258                         */
9259                        opt[IPOPT_OLEN] = (uint8_t)off;
9260                        while (off < optlen) {
9261                                opt[off++] = IPOPT_NOP;
9262                        }
9263                        break;
9264                case IPOPT_RR:
9265                        off = opt[IPOPT_OFFSET];
9266                        off--;
9267                        if (optlen < IP_ADDR_LEN ||
9268                            off > optlen - IP_ADDR_LEN) {
9269                                /* No more room - ignore */
9270                                ip1dbg((
9271                                    "ip_input_local_options: end of RR\n"));
9272                                break;
9273                        }
9274                        /* Pick a reasonable address on the outbound if */
9275                        if (ip_select_source_v4(ill, INADDR_ANY, ipha->ipha_dst,
9276                            INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9277                            NULL) != 0) {
9278                                /* No source! Shouldn't happen */
9279                                ifaddr = INADDR_ANY;
9280                        }
9281                        bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9282                        opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9283                        break;
9284                case IPOPT_TS:
9285                        /* Insert timestamp if there is romm */
9286                        switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9287                        case IPOPT_TS_TSONLY:
9288                                off = IPOPT_TS_TIMELEN;
9289                                break;
9290                        case IPOPT_TS_PRESPEC:
9291                        case IPOPT_TS_PRESPEC_RFC791:
9292                                /* Verify that the address matched */
9293                                off = opt[IPOPT_OFFSET] - 1;
9294                                bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9295                                if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9296                                        /* Not for us */
9297                                        break;
9298                                }
9299                                /* FALLTHRU */
9300                        case IPOPT_TS_TSANDADDR:
9301                                off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
```



```
9302                                if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9303                                        /* Not for us */
9304                                        break;
9305                                }
9306                                /* FALLTHRU */
9307                        default:
9308                                /*
9309                                 * ip_*put_options should have already
9310                                 * dropped this packet.
9311                                 */
9312                                cmn_err(CE_PANIC, "ip_input_local_options: "
9313                                    "unknown IT - bug in ip_input_options?\n");
9314                                return (B_TRUE);        /* Keep "lint" happy */
9315                        }
9316                        if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
9317                                /* Increase overflow counter */
9318                                off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
9319                                opt[IPOPT_POS_OV_FLG] =
9320                                    (uint8_t)((opt[IPOPT_POS_OV_FLG] & 0x0F) |
9321                                    (off << 4));
9322                                break;
9323                        }
9324                        off = opt[IPOPT_OFFSET] - 1;
9325                        switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9326                        case IPOPT_TS_PRESPEC:
9327                        case IPOPT_TS_PRESPEC_RFC791:
9328                        case IPOPT_TS_TSANDADDR:
9329                                /* Pick a reasonable addr on the outbound if */
9330                                if (ip_select_source_v4(ill, INADDR_ANY,
9331                                    ipha->ipha_dst, INADDR_ANY, ALL_ZONES, ipst,
9332                                    &ifaddr, NULL, NULL) != 0) {
9333                                        /* No source! Shouldn't happen */
9334                                        ifaddr = INADDR_ANY;
9335                                }
9336                                bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9337                                opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9338                                /* FALLTHRU */
9339                        case IPOPT_TS_TSONLY:
9340                                off = opt[IPOPT_OFFSET] - 1;
9341                                /* Compute # of milliseconds since midnight */
9342                                gethrestime(&now);
9343                                ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
9344                                    now.tv_nsec / (NANOSEC / MILLISEC);
9345                                bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
9346                                opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
9347                                break;
9348                        }
9349                        break;
9350                }
9351        }
9352        return (B_TRUE);

9353 bad_src_route:
9354        /* make sure we clear any indication of a hardware checksum */
9355        DB_CKSUMFLAGS(mp) = 0;
9356        ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
9357        icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
9358        return (B_FALSE);

9359 }

9360 /*
9361  * Process IP options in an inbound packet.  Always returns the nexthop.
9362  * Normally this is the passed in nexthop, but if there is an option
9363  * that effects the nexthop (such as a source route) that will be returned.
9364  * Sets *errorp if there is an error, in which case an ICMP error has been sent
9365  * and mp freed.
9366  */
9367 ipaddr_t
9368 ip_input_options(ipha_t *ipha, ipaddr_t dst, mblk_t *mp,
```

```
9368     ip_recv_attr_t *ira, int *errorp)
9369 {
9370         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
9371         ipoptp_t        opts;
9372         uchar_t         *opt;
9373         uint8_t         optval;
9374         uint8_t         optlen;
9375         intptr_t        code = 0;
9376         ire_t           *ire;

9378         ip2dbg(("ip_input_options\n"));
9379         *errorp = 0;
9380         for (optval = ipoptp_first(&opts, ipha);
9381             optval != IPOPT_EOL;
9382             optval = ipoptp_next(&opts)) {
9383                 opt = opts.ipoptp_cur;
9384                 optlen = opts.ipoptp_len;
9385                 ip2dbg(("ip_input_options: opt %d, len %d\n",
9386                     optval, optlen));
9387                 /*
9388                  * Note: we need to verify the checksum before we
9389                  * modify anything thus this routine only extracts the next
9390                  * hop dst from any source route.
9391                  */
9392                 switch (optval) {
9393                         uint32_t off;
9394                 case IPOPT_SSRR:
9395                 case IPOPT_LSRR:
9396                         if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9397                                 if (optval == IPOPT_SSRR) {
9398                                         ip1dbg(("ip_input_options: not next"
9399                                             " strict source route 0x%x\n",
9400                                             ntohl(dst)));
9401                                         code = (char *)&ipha->ipha_dst -
9402                                             (char *)ipha;
9403                                         goto param_prob; /* RouterReq's */
9404                                 }
9405                                 ip2dbg(("ip_input_options: "
9406                                     "not next source route 0x%x\n",
9407                                     ntohl(dst)));
9408                                 break;
9409                         }

9411                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9412                                 ip1dbg((
9413                                     "ip_input_options: bad option offset\n"));
9414                                 code = (char *)&opt[IPOPT_OLEN] -
9415                                     (char *)ipha;
9416                                 goto param_prob;
9417                         }
9418                         off = opt[IPOPT_OFFSET];
9419                         off--;
9420                 redo_srr:
9421                         if (optlen < IP_ADDR_LEN ||
9422                             off > optlen - IP_ADDR_LEN) {
9423                                 /* End of source route */
9424                                 ip1dbg(("ip_input_options: end of SR\n"));
9425                                 break;
9426                         }
9427                         bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9428                         ip1dbg(("ip_input_options: next hop 0x%x\n",
9429                             ntohl(dst)));

9431                         /*
9432                          * Check if our address is present more than
9433                          * once as consecutive hops in source route.
```



```
9434                          * XXX verify per-interface ip_forwarding
9435                          * for source route?
9436                          */
9437                         if (ip_type_v4(dst, ipst) == IRE_LOCAL) {
9438                                 off += IP_ADDR_LEN;
9439                                 goto redo_srr;
9440                         }

9442                         if (dst == htonl(INADDR_LOOPBACK)) {
9443                                 ip1dbg(("ip_input_options: loopback addr in "
9444                                     "source route!\n"));
9445                                 goto bad_src_route;
9446                         }
9447                         /*
9448                          * For strict: verify that dst is directly
9449                          * reachable.
9450                          */
9451                         if (optval == IPOPT_SSRR) {
9452                                 ire = ire_ftable_lookup_v4(dst, 0, 0,
9453                                     IRE_INTERFACE, NULL, ALL_ZONES,
9454                                     ira->ira_tsl,
9455                                     MATCH_IRE_TYPE | MATCH_IRE_SECATTR, 0, ipst,
9456                                     NULL);
9457                                 if (ire == NULL) {
9458                                         ip1dbg(("ip_input_options: SSRR not "
9459                                             "directly reachable: 0x%x\n",
9460                                             ntohl(dst)));
9461                                         goto bad_src_route;
9462                                 }
9463                                 ire_refrele(ire);
9464                         }
9465                         /*
9466                          * Defer update of the offset and the record route
9467                          * until the packet is forwarded.
9468                          */
9469                         break;
9470                 case IPOPT_RR:
9471                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9472                                 ip1dbg((
9473                                     "ip_input_options: bad option offset\n"));
9474                                 code = (char *)&opt[IPOPT_OLEN] -
9475                                     (char *)ipha;
9476                                 goto param_prob;
9477                         }
9478                         break;
9479                 case IPOPT_TS:
9480                         /*
9481                          * Verify that length >= 5 and that there is either
9482                          * room for another timestamp or that the overflow
9483                          * counter is not maxed out.
9484                          */
9485                         code = (char *)&opt[IPOPT_OLEN] - (char *)ipha;
9486                         if (optlen < IPOPT_MINLEN_IT) {
9487                                 goto param_prob;
9488                         }
9489                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9490                                 ip1dbg((
9491                                     "ip_input_options: bad option offset\n"));
9492                                 code = (char *)&opt[IPOPT_OFFSET] -
9493                                     (char *)ipha;
9494                                 goto param_prob;
9495                         }
9496                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9497                         case IPOPT_TS_TSONLY:
9498                                 off = IPOPT_TS_TIMELEN;
9499                                 break;
```

```
9500                        case IPOPT_TS_TSANDADDR:
9501                        case IPOPT_TS_PRESPEC:
9502                        case IPOPT_TS_PRESPEC_RFC791:
9503                                off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
9504                                break;
9505                        default:
9506                                code = (char *)&opt[IPOPT_POS_OV_FLG] -
9507                                    (char *)ipha;
9508                                goto param_prob;
9509                        }
9510                        if (opt[IPOPT_OFFSET] - 1 + off > optlen &&
9511                            (opt[IPOPT_POS_OV_FLG] & 0xF0) == 0xF0) {
9512                                /*
9513                                 * No room and the overflow counter is 15
9514                                 * already.
9515                                 */
9516                                goto param_prob;
9517                        }
9518                        break;
9519                }
9520        }

9522        if ((opts.ipoptp_flags & IPOPTP_ERROR) == 0) {
9523                return (dst);
9524        }

9526        ip1dbg(("ip_input_options: error processing IP options."));
9527        code = (char *)&opt[IPOPT_OFFSET] - (char *)ipha;

9529 param_prob:
9530        /* make sure we clear any indication of a hardware checksum */
9531        DB_CKSUMFLAGS(mp) = 0;
9532        ip_drop_input("ICMP_PARAM_PROBLEM", mp, ira->ira_ill);
9533        icmp_param_problem(mp, (uint8_t)code, ira);
9534        *errorp = -1;
9535        return (dst);

9537 bad_src_route:
9538        /* make sure we clear any indication of a hardware checksum */
9539        DB_CKSUMFLAGS(mp) = 0;
9540        ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ira->ira_ill);
9541        icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
9542        *errorp = -1;
9543        return (dst);
9544 }

9546 /*
9547  * IP & ICMP info in >=14 msg's ...
9548  *  - ip fixed part (mib2_ip_t)
9549  *  - icmp fixed part (mib2_icmp_t)
9550  *  - ipAddrEntryTable (ip 20)          all IPv4 ipifs
9551  *  - ipRouteEntryTable (ip 21)         all IPv4 IREs
9552  *  - ipNetToMediaEntryTable (ip 22)    all IPv4 Neighbor Cache entries
9553  *  - ipRouteAttributeTable (ip 102)    labeled routes
9554  *  - ip multicast membership (ip_member_t)
9555  *  - ip multicast source filtering (ip_grpsrc_t)
9556  *  - igmp fixed part (struct igmpstat)
9557  *  - multicast routing stats (struct mrtstat)
9558  *  - multicast routing vifs (array of struct vifctl)
9559  *  - multicast routing routes (array of struct mfcctl)
9560  *  - ip6 fixed part (mib2_ipv6IfStatsEntry_t)
9561  *                                      One per ill plus one generic
9562  *  - icmp6 fixed part (mib2_ipv6IfIcmpEntry_t)
9563  *                                      One per ill plus one generic
9564  *  - ipv6RouteEntry                    all IPv6 IREs
9565  *  - ipv6RouteAttributeTable (ip6 102) labeled routes
```



```
9566  *  - ipv6NetToMediaEntry               all IPv6 Neighbor Cache entries
9567  *  - ipv6AddrEntry                     all IPv6 ipifs
9568  *  - ipv6 multicast membership (ipv6_member_t)
9569  *  - ipv6 multicast source filtering (ipv6_grpsrc_t)
9570  *
9571  * NOTE: original mpctl is copied for msg's 2..N, since its ctl part is
9572  * already filled in by the caller.
9573  * If legacy_req is true then MIB structures needs to be truncated to their
9574  * legacy sizes before being returned.
9575  * Return value of 0 indicates that no messages were sent and caller
9576  * should free mpctl.
9577  */
9578 int
9579 ip_snmp_get(queue_t *q, mblk_t *mpctl, int level, boolean_t legacy_req)
9580 {
9581         ip_stack_t *ipst;
9582         sctp_stack_t *sctps;

9584         if (q->q_next != NULL) {
9585                 ipst = ILLQ_TO_IPST(q);
9586         } else {
9587                 ipst = CONNQ_TO_IPST(q);
9588         }
9589         ASSERT(ipst != NULL);
9590         sctps = ipst->ips_netstack->netstack_sctp;

9592         if (mpctl == NULL || mpctl->b_cont == NULL) {
9593                 return (0);
9594         }

9596         /*
9597          * For the purposes of the (broken) packet shell use
9598          * of the level we make sure MIB2_TCP/MIB2_UDP can be used
9599          * to make TCP and UDP appear first in the list of mib items.
9600          * TBD: We could expand this and use it in netstat so that
9601          * the kernel doesn't have to produce large tables (connections,
9602          * routes, etc) when netstat only wants the statistics or a particular
9603          * table.
9604          */
9605         if (!(level == MIB2_TCP || level == MIB2_UDP)) {
9606                 if ((mpctl = icmp_snmp_get(q, mpctl)) == NULL) {
9607                         return (1);
9608                 }
9609         }

9611         if (level != MIB2_TCP) {
9612                 if ((mpctl = udp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9613                         return (1);
9614                 }
9615         }

9617         if (level != MIB2_UDP) {
9618                 if ((mpctl = tcp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9619                         return (1);
9620                 }
9621         }

9623         if ((mpctl = ip_snmp_get_mib2_ip_traffic_stats(q, mpctl,
9624             ipst, legacy_req)) == NULL) {
9625                 return (1);
9626         }

9628         if ((mpctl = ip_snmp_get_mib2_ip6(q, mpctl, ipst,
9629             legacy_req)) == NULL) {
9630                 return (1);
9631         }
```

```
9633         if ((mpctl = ip_snmp_get_mib2_icmp(q, mpctl, ipst)) == NULL) {
9634                 return (1);
9635         }

9637         if ((mpctl = ip_snmp_get_mib2_icmp6(q, mpctl, ipst)) == NULL) {
9638                 return (1);
9639         }

9641         if ((mpctl = ip_snmp_get_mib2_igmp(q, mpctl, ipst)) == NULL) {
9642                 return (1);
9643         }

9645         if ((mpctl = ip_snmp_get_mib2_multi(q, mpctl, ipst)) == NULL) {
9646                 return (1);
9647         }

9649         if ((mpctl = ip_snmp_get_mib2_ip_addr(q, mpctl, ipst,
9650             legacy_req)) == NULL) {
9651                 return (1);
9652         }

9654         if ((mpctl = ip_snmp_get_mib2_ip6_addr(q, mpctl, ipst,
9655             legacy_req)) == NULL) {
9656                 return (1);
9657         }

9659         if ((mpctl = ip_snmp_get_mib2_ip_group_mem(q, mpctl, ipst)) == NULL) {
9660                 return (1);
9661         }

9663         if ((mpctl = ip_snmp_get_mib2_ip6_group_mem(q, mpctl, ipst)) == NULL) {
9664                 return (1);
9665         }

9667         if ((mpctl = ip_snmp_get_mib2_ip_group_src(q, mpctl, ipst)) == NULL) {
9668                 return (1);
9669         }

9671         if ((mpctl = ip_snmp_get_mib2_ip6_group_src(q, mpctl, ipst)) == NULL) {
9672                 return (1);
9673         }

9675         if ((mpctl = ip_snmp_get_mib2_virt_multi(q, mpctl, ipst)) == NULL) {
9676                 return (1);
9677         }

9679         if ((mpctl = ip_snmp_get_mib2_multi_rtable(q, mpctl, ipst)) == NULL) {
9680                 return (1);
9681         }

9683         mpctl = ip_snmp_get_mib2_ip_route_media(q, mpctl, level, ipst);
9684         if (mpctl == NULL)
9685                 return (1);

9687         mpctl = ip_snmp_get_mib2_ip6_route_media(q, mpctl, level, ipst);
9688         if (mpctl == NULL)
9689                 return (1);

9691         if ((mpctl = sctp_snmp_get_mib2(q, mpctl, sctps)) == NULL) {
9692                 return (1);
9693         }

9695 #endif /* ! codereview */
9696         if ((mpctl = ip_snmp_get_mib2_ip_dce(q, mpctl, ipst)) == NULL) {
9697                 return (1);
```



```
9698         }

9700         if ((mpctl = dccp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9701                 return (1);
9702         }

9704 #endif /* ! codereview */
9705         freemsg(mpctl);
9706         return (1);
9707 }

9709 /* Get global (legacy) IPv4 statistics */
9710 static mblk_t *
9711 ip_snmp_get_mib2_ip(queue_t *q, mblk_t *mpctl, mib2_ipIfStatsEntry_t *ipmib,
9712     ip_stack_t *ipst, boolean_t legacy_req)
9713 {
9714         mib2_ip_t               old_ip_mib;
9715         struct opthdr           *optp;
9716         mblk_t                  *mp2ctl;
9717         mib2_ipAddrEntry_t      mae;

9719         /*
9720          * make a copy of the original message
9721          */
9722         mp2ctl = copymsg(mpctl);

9724         /* fixed length IP structure... */
9725         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9726         optp->level = MIB2_IP;
9727         optp->name = 0;
9728         SET_MIB(old_ip_mib.ipForwarding,
9729             (WE_ARE_FORWARDING(ipst) ? 1 : 2));
9730         SET_MIB(old_ip_mib.ipDefaultTTL,
9731             (uint32_t)ipst->ips_ip_def_ttl);
9732         SET_MIB(old_ip_mib.ipReasmTimeout,
9733             ipst->ips_ip_reassembly_timeout);
9734         SET_MIB(old_ip_mib.ipAddrEntrySize,
9735             (legacy_req) ? LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t) :
9736             sizeof (mib2_ipAddrEntry_t));
9737         SET_MIB(old_ip_mib.ipRouteEntrySize,
9738             sizeof (mib2_ipRouteEntry_t));
9739         SET_MIB(old_ip_mib.ipNetToMediaEntrySize,
9740             sizeof (mib2_ipNetToMediaEntry_t));
9741         SET_MIB(old_ip_mib.ipMemberEntrySize, sizeof (ip_member_t));
9742         SET_MIB(old_ip_mib.ipGroupSourceEntrySize, sizeof (ip_grpsrc_t));
9743         SET_MIB(old_ip_mib.ipRouteAttributeSize,
9744             sizeof (mib2_ipAttributeEntry_t));
9745         SET_MIB(old_ip_mib.transportMLPSize, sizeof (mib2_transportMLPEntry_t));
9746         SET_MIB(old_ip_mib.ipDestEntrySize, sizeof (dest_cache_entry_t));

9748         /*
9749          * Grab the statistics from the new IP MIB
9750          */
9751         SET_MIB(old_ip_mib.ipInReceives,
9752             (uint32_t)ipmib->ipIfStatsHCInReceives);
9753         SET_MIB(old_ip_mib.ipInHdrErrors, ipmib->ipIfStatsInHdrErrors);
9754         SET_MIB(old_ip_mib.ipInAddrErrors, ipmib->ipIfStatsInAddrErrors);
9755         SET_MIB(old_ip_mib.ipForwDatagrams,
9756             (uint32_t)ipmib->ipIfStatsHCOutForwDatagrams);
9757         SET_MIB(old_ip_mib.ipInUnknownProtos,
9758             ipmib->ipIfStatsInUnknownProtos);
9759         SET_MIB(old_ip_mib.ipInDiscards, ipmib->ipIfStatsInDiscards);
9760         SET_MIB(old_ip_mib.ipInDelivers,
9761             (uint32_t)ipmib->ipIfStatsHCInDelivers);
9762         SET_MIB(old_ip_mib.ipOutRequests,
9763             (uint32_t)ipmib->ipIfStatsHCOutRequests);
```

```
9764         SET_MIB(old_ip_mib.ipOutDiscards, ipmib->ipIfStatsOutDiscards);
9765         SET_MIB(old_ip_mib.ipOutNoRoutes, ipmib->ipIfStatsOutNoRoutes);
9766         SET_MIB(old_ip_mib.ipReasmReqds, ipmib->ipIfStatsReasmReqds);
9767         SET_MIB(old_ip_mib.ipReasmOKs, ipmib->ipIfStatsReasmOKs);
9768         SET_MIB(old_ip_mib.ipReasmFails, ipmib->ipIfStatsReasmFails);
9769         SET_MIB(old_ip_mib.ipFragOKs, ipmib->ipIfStatsOutFragOKs);
9770         SET_MIB(old_ip_mib.ipFragFails, ipmib->ipIfStatsOutFragFails);
9771         SET_MIB(old_ip_mib.ipFragCreates, ipmib->ipIfStatsOutFragCreates);

9773         /* ipRoutingDiscards is not being used */
9774         SET_MIB(old_ip_mib.ipRoutingDiscards, 0);
9775         SET_MIB(old_ip_mib.tcpInErrs, ipmib->tcpIfStatsInErrs);
9776         SET_MIB(old_ip_mib.udpNoPorts, ipmib->udpIfStatsNoPorts);
9777         SET_MIB(old_ip_mib.ipInCksumErrs, ipmib->ipIfStatsInCksumErrs);
9778         SET_MIB(old_ip_mib.ipReasmDuplicates,
9779             ipmib->ipIfStatsReasmDuplicates);
9780         SET_MIB(old_ip_mib.ipReasmPartDups, ipmib->ipIfStatsReasmPartDups);
9781         SET_MIB(old_ip_mib.ipForwProhibits, ipmib->ipIfStatsForwProhibits);
9782         SET_MIB(old_ip_mib.udpInCksumErrs, ipmib->udpIfStatsInCksumErrs);
9783         SET_MIB(old_ip_mib.udpInOverflows, ipmib->udpIfStatsInOverflows);
9784         SET_MIB(old_ip_mib.rawipInOverflows,
9785             ipmib->rawipIfStatsInOverflows);

9787         SET_MIB(old_ip_mib.ipsecInSucceeded, ipmib->ipsecIfStatsInSucceeded);
9788         SET_MIB(old_ip_mib.ipsecInFailed, ipmib->ipsecIfStatsInFailed);
9789         SET_MIB(old_ip_mib.ipInIPv6, ipmib->ipIfStatsInWrongIPVersion);
9790         SET_MIB(old_ip_mib.ipOutIPv6, ipmib->ipIfStatsOutWrongIPVersion);
9791         SET_MIB(old_ip_mib.ipOutSwitchIPv6,
9792             ipmib->ipIfStatsOutSwitchIPVersion);

9794         if (!snmp_append_data(mpctl->b_cont, (char *)&old_ip_mib,
9795             (int)sizeof (old_ip_mib))) {
9796                 ip1dbg(("ip_snmp_get_mib2_ip: failed to allocate %u bytes\n",
9797                     (uint_t)sizeof (old_ip_mib)));
9798         }

9800         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9801         ip3dbg(("ip_snmp_get_mib2_ip: level %d, name %d, len %d\n",
9802             (int)optp->level, (int)optp->name, (int)optp->len));
9803         qreply(q, mpctl);
9804         return (mp2ctl);
9805 }

9807 /* Per interface IPv4 statistics */
9808 static mblk_t *
9809 ip_snmp_get_mib2_ip_traffic_stats(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
9810     boolean_t legacy_req)
9811 {
9812         struct opthdr           *optp;
9813         mblk_t                  *mp2ctl;
9814         ill_t                   *ill;
9815         ill_walk_context_t      ctx;
9816         mblk_t                  *mp_tail = NULL;
9817         mib2_ipIfStatsEntry_t   global_ip_mib;
9818         mib2_ipAddrEntry_t      mae;

9820         /*
9821          * Make a copy of the original message
9822          */
9823         mp2ctl = copymsg(mpctl);

9825         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9826         optp->level = MIB2_IP;
9827         optp->name = MIB2_IP_TRAFFIC_STATS;
9828         /* Include "unknown interface" ip_mib */
9829         ipst->ips_ip_mib.ipIfStatsIPVersion = MIB2_INETADDRESSTYPE_ipv4;
```



```
9830         ipst->ips_ip_mib.ipIfStatsIfIndex =
9831             MIB2_UNKNOWN_INTERFACE; /* Flag to netstat */
9832         SET_MIB(ipst->ips_ip_mib.ipIfStatsForwarding,
9833             (ipst->ips_ip_forwarding ? 1 : 2));
9834         SET_MIB(ipst->ips_ip_mib.ipIfStatsDefaultTTL,
9835             (uint32_t)ipst->ips_ip_def_ttl);
9836         SET_MIB(ipst->ips_ip_mib.ipIfStatsEntrySize,
9837             sizeof (mib2_ipIfStatsEntry_t));
9838         SET_MIB(ipst->ips_ip_mib.ipIfStatsAddrEntrySize,
9839             sizeof (mib2_ipAddrEntry_t));
9840         SET_MIB(ipst->ips_ip_mib.ipIfStatsRouteEntrySize,
9841             sizeof (mib2_ipRouteEntry_t));
9842         SET_MIB(ipst->ips_ip_mib.ipIfStatsNetToMediaEntrySize,
9843             sizeof (mib2_ipNetToMediaEntry_t));
9844         SET_MIB(ipst->ips_ip_mib.ipIfStatsMemberEntrySize,
9845             sizeof (ip_member_t));
9846         SET_MIB(ipst->ips_ip_mib.ipIfStatsGroupSourceEntrySize,
9847             sizeof (ip_grpsrc_t));

9849         bcopy(&ipst->ips_ip_mib, &global_ip_mib, sizeof (global_ip_mib));

9851         if (legacy_req) {
9852                 SET_MIB(global_ip_mib.ipIfStatsAddrEntrySize,
9853                     LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t));
9854         }

9856         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
9857             (char *)&global_ip_mib, (int)sizeof (global_ip_mib))) {
9858                 ip1dbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9859                     "failed to allocate %u bytes\n",
9860                     (uint_t)sizeof (global_ip_mib)));
9861         }

9863         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
9864         ill = ILL_START_WALK_V4(&ctx, ipst);
9865         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
9866                 ill->ill_ip_mib->ipIfStatsIfIndex =
9867                     ill->ill_phyint->phyint_ifindex;
9868                 SET_MIB(ill->ill_ip_mib->ipIfStatsForwarding,
9869                     (ipst->ips_ip_forwarding ? 1 : 2));
9870                 SET_MIB(ill->ill_ip_mib->ipIfStatsDefaultTTL,
9871                     (uint32_t)ipst->ips_ip_def_ttl);

9873                 ip_mib2_add_ip_stats(&global_ip_mib, ill->ill_ip_mib);
9874                 if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
9875                     (char *)ill->ill_ip_mib,
9876                     (int)sizeof (*ill->ill_ip_mib))) {
9877                         ip1dbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9878                             "failed to allocate %u bytes\n",
9879                             (uint_t)sizeof (*ill->ill_ip_mib)));
9880                 }
9881         }
9882         rw_exit(&ipst->ips_ill_g_lock);

9884         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9885         ip3dbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9886             "level %d, name %d, len %d\n",
9887             (int)optp->level, (int)optp->name, (int)optp->len));
9888         qreply(q, mpctl);

9890         if (mp2ctl == NULL)
9891                 return (NULL);

9893         return (ip_snmp_get_mib2_ip(q, mp2ctl, &global_ip_mib, ipst,
9894             legacy_req));
9895 }
```

```
9897 /* Global IPv4 ICMP statistics */
9898 static mblk_t *
9899 ip_snmp_get_mib2_icmp(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9900 {
9901         struct opthdr           *optp;
9902         mblk_t                  *mp2ctl;

9904         /*
9905          * Make a copy of the original message
9906          */
9907         mp2ctl = copymsg(mpctl);

9909         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9910         optp->level = MIB2_ICMP;
9911         optp->name = 0;
9912         if (!snmp_append_data(mpctl->b_cont, (char *)&ipst->ips_icmp_mib,
9913             (int)sizeof (ipst->ips_icmp_mib))) {
9914                 ip1dbg(("ip_snmp_get_mib2_icmp: failed to allocate %u bytes\n",
9915                     (uint_t)sizeof (ipst->ips_icmp_mib)));
9916         }
9917         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9918         ip3dbg(("ip_snmp_get_mib2_icmp: level %d, name %d, len %d\n",
9919             (int)optp->level, (int)optp->name, (int)optp->len));
9920         qreply(q, mpctl);
9921         return (mp2ctl);
9922 }

9924 /* Global IPv4 IGMP statistics */
9925 static mblk_t *
9926 ip_snmp_get_mib2_igmp(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9927 {
9928         struct opthdr           *optp;
9929         mblk_t                  *mp2ctl;

9931         /*
9932          * make a copy of the original message
9933          */
9934         mp2ctl = copymsg(mpctl);

9936         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9937         optp->level = EXPER_IGMP;
9938         optp->name = 0;
9939         if (!snmp_append_data(mpctl->b_cont, (char *)&ipst->ips_igmpstat,
9940             (int)sizeof (ipst->ips_igmpstat))) {
9941                 ip1dbg(("ip_snmp_get_mib2_igmp: failed to allocate %u bytes\n",
9942                     (uint_t)sizeof (ipst->ips_igmpstat)));
9943         }
9944         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9945         ip3dbg(("ip_snmp_get_mib2_igmp: level %d, name %d, len %d\n",
9946             (int)optp->level, (int)optp->name, (int)optp->len));
9947         qreply(q, mpctl);
9948         return (mp2ctl);
9949 }

9951 /* Global IPv4 Multicast Routing statistics */
9952 static mblk_t *
9953 ip_snmp_get_mib2_multi(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9954 {
9955         struct opthdr           *optp;
9956         mblk_t                  *mp2ctl;

9958         /*
9959          * make a copy of the original message
9960          */
9961         mp2ctl = copymsg(mpctl);
```



```
9963         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9964         optp->level = EXPER_DVMRP;
9965         optp->name = 0;
9966         if (!ip_mroute_stats(mpctl->b_cont, ipst)) {
9967                 ip0dbg(("ip_mroute_stats: failed\n"));
9968         }
9969         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9970         ip3dbg(("ip_snmp_get_mib2_multi: level %d, name %d, len %d\n",
9971             (int)optp->level, (int)optp->name, (int)optp->len));
9972         qreply(q, mpctl);
9973         return (mp2ctl);
9974 }

9976 /* IPv4 address information */
9977 static mblk_t *
9978 ip_snmp_get_mib2_ip_addr(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
9979     boolean_t legacy_req)
9980 {
9981         struct opthdr           *optp;
9982         mblk_t                  *mp2ctl;
9983         mblk_t                  *mp_tail = NULL;
9984         ill_t                   *ill;
9985         ipif_t                  *ipif;
9986         uint_t                  bitval;
9987         mib2_ipAddrEntry_t      mae;
9988         size_t                  mae_size;
9989         zoneid_t                zoneid;
9990         ill_walk_context_t      ctx;

9992         /*
9993          * make a copy of the original message
9994          */
9995         mp2ctl = copymsg(mpctl);

9997         mae_size = (legacy_req) ? LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t) :
9998             sizeof (mib2_ipAddrEntry_t);

10000         /* ipAddrEntryTable */

10002         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10003         optp->level = MIB2_IP;
10004         optp->name = MIB2_IP_ADDR;
10005         zoneid = Q_TO_CONN(q)->conn_zoneid;

10007         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10008         ill = ILL_START_WALK_V4(&ctx, ipst);
10009         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10010                 for (ipif = ill->ill_ipif; ipif != NULL;
10011                     ipif = ipif->ipif_next) {
10012                         if (ipif->ipif_zoneid != zoneid &&
10013                             ipif->ipif_zoneid != ALL_ZONES)
10014                                 continue;
10015                         /* Sum of count from dead IRE_LO* and our current */
10016                         mae.ipAdEntInfo.ae_ibcnt = ipif->ipif_ib_pkt_count;
10017                         if (ipif->ipif_ire_local != NULL) {
10018                                 mae.ipAdEntInfo.ae_ibcnt +=
10019                                     ipif->ipif_ire_local->ire_ib_pkt_count;
10020                         }
10021                         mae.ipAdEntInfo.ae_obcnt = 0;
10022                         mae.ipAdEntInfo.ae_focnt = 0;

10024                         ipif_get_name(ipif, mae.ipAdEntIfIndex.o_bytes,
10025                             OCTET_LENGTH);
10026                         mae.ipAdEntIfIndex.o_length =
10027                             mi_strlen(mae.ipAdEntIfIndex.o_bytes);
```

```
10028                       mae.ipAdEntAddr = ipif->ipif_lcl_addr;
10029                       mae.ipAdEntNetMask = ipif->ipif_net_mask;
10030                       mae.ipAdEntInfo.ae_subnet = ipif->ipif_subnet;
10031                       mae.ipAdEntInfo.ae_subnet_len =
10032                           ip_mask_to_plen(ipif->ipif_net_mask);
10033                       mae.ipAdEntInfo.ae_src_addr = ipif->ipif_lcl_addr;
10034                       for (bitval = 1;
10035                           bitval &&
10036                           !(bitval & ipif->ipif_brd_addr);
10037                           bitval <<= 1)
10038                               noop;
10039                       mae.ipAdEntBcastAddr = bitval;
10040                       mae.ipAdEntReasmMaxSize = IP_MAXPACKET;
10041                       mae.ipAdEntInfo.ae_mtu = ipif->ipif_ill->ill_mtu;
10042                       mae.ipAdEntInfo.ae_metric  = ipif->ipif_ill->ill_metric;
10043                       mae.ipAdEntInfo.ae_broadcast_addr =
10044                           ipif->ipif_brd_addr;
10045                       mae.ipAdEntInfo.ae_pp_dst_addr =
10046                           ipif->ipif_pp_dst_addr;
10047                       mae.ipAdEntInfo.ae_flags = ipif->ipif_flags |
10048                           ill->ill_flags | ill->ill_phyint->phyint_flags;
10049                       mae.ipAdEntRetransmitTime =
10050                           ill->ill_reachable_retrans_time;

10052                       if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10053                           (char *)&mae, (int)mae_size)) {
10054                               ip1dbg(("ip_snmp_get_mib2_ip_addr: failed to "
10055                                   "allocate %u bytes\n", (uint_t)mae_size));
10056                       }
10057               }
10058       }
10059       rw_exit(&ipst->ips_ill_g_lock);

10061       optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10062       ip3dbg(("ip_snmp_get_mib2_ip_addr: level %d, name %d, len %d\n",
10063           (int)optp->level, (int)optp->name, (int)optp->len));
10064       qreply(q, mpctl);
10065       return (mp2ctl);
10066 }

10068 /* IPv6 address information */
10069 static mblk_t *
10070 ip_snmp_get_mib2_ip6_addr(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
10071     boolean_t legacy_req)
10072 {
10073       struct opthdr           *optp;
10074       mblk_t                  *mp2ctl;
10075       mblk_t                  *mp_tail = NULL;
10076       ill_t                   *ill;
10077       ipif_t                  *ipif;
10078       mib2_ipv6AddrEntry_t    mae6;
10079       size_t                  mae6_size;
10080       zoneid_t                zoneid;
10081       ill_walk_context_t      ctx;

10083       /*
10084        * make a copy of the original message
10085        */
10086       mp2ctl = copymsg(mpctl);

10088       mae6_size = (legacy_req) ?
10089           LEGACY_MIB_SIZE(&mae6, mib2_ipv6AddrEntry_t) :
10090           sizeof (mib2_ipv6AddrEntry_t);

10092       /* ipv6AddrEntryTable */
```



```
10094       optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10095       optp->level = MIB2_IP6;
10096       optp->name = MIB2_IP6_ADDR;
10097       zoneid = Q_TO_CONN(q)->conn_zoneid;

10099       rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10100       ill = ILL_START_WALK_V6(&ctx, ipst);
10101       for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10102               for (ipif = ill->ill_ipif; ipif != NULL;
10103                   ipif = ipif->ipif_next) {
10104                       if (ipif->ipif_zoneid != zoneid &&
10105                           ipif->ipif_zoneid != ALL_ZONES)
10106                               continue;
10107                       /* Sum of count from dead IRE_LO* and our current */
10108                       mae6.ipv6AddrInfo.ae_ibcnt = ipif->ipif_ib_pkt_count;
10109                       if (ipif->ipif_ire_local != NULL) {
10110                               mae6.ipv6AddrInfo.ae_ibcnt +=
10111                                   ipif->ipif_ire_local->ire_ib_pkt_count;
10112                       }
10113                       mae6.ipv6AddrInfo.ae_obcnt = 0;
10114                       mae6.ipv6AddrInfo.ae_focnt = 0;

10116                       ipif_get_name(ipif, mae6.ipv6AddrIfIndex.o_bytes,
10117                           OCTET_LENGTH);
10118                       mae6.ipv6AddrIfIndex.o_length =
10119                           mi_strlen(mae6.ipv6AddrIfIndex.o_bytes);
10120                       mae6.ipv6AddrAddress = ipif->ipif_v6lcl_addr;
10121                       mae6.ipv6AddrPfxLength =
10122                           ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
10123                       mae6.ipv6AddrInfo.ae_subnet = ipif->ipif_v6subnet;
10124                       mae6.ipv6AddrInfo.ae_subnet_len =
10125                           mae6.ipv6AddrPfxLength;
10126                       mae6.ipv6AddrInfo.ae_src_addr = ipif->ipif_v6lcl_addr;

10128                       /* Type: stateless(1), stateful(2), unknown(3) */
10129                       if (ipif->ipif_flags & IPIF_ADDRCONF)
10130                               mae6.ipv6AddrType = 1;
10131                       else
10132                               mae6.ipv6AddrType = 2;
10133                       /* Anycast: true(1), false(2) */
10134                       if (ipif->ipif_flags & IPIF_ANYCAST)
10135                               mae6.ipv6AddrAnycastFlag = 1;
10136                       else
10137                               mae6.ipv6AddrAnycastFlag = 2;

10139                       /*
10140                        * Address status: preferred(1), deprecated(2),
10141                        * invalid(3), inaccessible(4), unknown(5)
10142                        */
10143                       if (ipif->ipif_flags & IPIF_NOLOCAL)
10144                               mae6.ipv6AddrStatus = 3;
10145                       else if (ipif->ipif_flags & IPIF_DEPRECATED)
10146                               mae6.ipv6AddrStatus = 2;
10147                       else
10148                               mae6.ipv6AddrStatus = 1;
10149                       mae6.ipv6AddrInfo.ae_mtu = ipif->ipif_ill->ill_mtu;
10150                       mae6.ipv6AddrInfo.ae_metric  =
10151                           ipif->ipif_ill->ill_metric;
10152                       mae6.ipv6AddrInfo.ae_pp_dst_addr =
10153                           ipif->ipif_v6pp_dst_addr;
10154                       mae6.ipv6AddrInfo.ae_flags = ipif->ipif_flags |
10155                           ill->ill_flags | ill->ill_phyint->phyint_flags;
10156                       mae6.ipv6AddrReasmMaxSize = IP_MAXPACKET;
10157                       mae6.ipv6AddrIdentifier = ill->ill_token;
10158                       mae6.ipv6AddrIdentifierLen = ill->ill_token_length;
10159                       mae6.ipv6AddrReachableTime = ill->ill_reachable_time;
```

```
10160                         mae6.ipv6AddrRetransmitTime =
10161                             ill->ill_reachable_retrans_time;
10162                         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10163                             (char *)&mae6, (int)mae6_size)) {
10164                             ip1dbg(("ip_snmp_get_mib2_ip6_addr: failed to "
10165                                 "allocate %u bytes\n",
10166                                 (uint_t)mae6_size));
10167                         }
10168                 }
10169         }
10170         rw_exit(&ipst->ips_ill_g_lock);

10172         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10173         ip3dbg(("ip_snmp_get_mib2_ip6_addr: level %d, name %d, len %d\n",
10174             (int)optp->level, (int)optp->name, (int)optp->len));
10175         qreply(q, mpctl);
10176         return (mp2ctl);
10177 }

10179 /* IPv4 multicast group membership. */
10180 static mblk_t *
10181 ip_snmp_get_mib2_ip_group_mem(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10182 {
10183         struct opthdr           *optp;
10184         mblk_t                  *mp2ctl;
10185         ill_t                   *ill;
10186         ipif_t                  *ipif;
10187         ilm_t                   *ilm;
10188         ip_member_t             ipm;
10189         mblk_t                  *mp_tail = NULL;
10190         ill_walk_context_t      ctx;
10191         zoneid_t                zoneid;

10193         /*
10194          * make a copy of the original message
10195          */
10196         mp2ctl = copymsg(mpctl);
10197         zoneid = Q_TO_CONN(q)->conn_zoneid;

10199         /* ipGroupMember table */
10200         optp = (struct opthdr *)&mpctl->b_rptr[
10201             sizeof (struct T_optmgmt_ack)];
10202         optp->level = MIB2_IP;
10203         optp->name = EXPER_IP_GROUP_MEMBERSHIP;

10205         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10206         ill = ILL_START_WALK_V4(&ctx, ipst);
10207         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10208                 /* Make sure the ill isn't going away. */
10209                 if (!ill_check_and_refhold(ill))
10210                         continue;
10211                 rw_exit(&ipst->ips_ill_g_lock);
10212                 rw_enter(&ill->ill_mcast_lock, RW_READER);
10213                 for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10214                         if (ilm->ilm_zoneid != zoneid &&
10215                             ilm->ilm_zoneid != ALL_ZONES)
10216                                 continue;

10218                         /* Is there an ipif for ilm_ifaddr? */
10219                         for (ipif = ill->ill_ipif; ipif != NULL;
10220                             ipif = ipif->ipif_next) {
10221                                 if (!IPIF_IS_CONDEMNED(ipif) &&
10222                                     ipif->ipif_lcl_addr == ilm->ilm_ifaddr &&
10223                                     ilm->ilm_ifaddr != INADDR_ANY)
10224                                         break;
10225                         }
```



```
10226                         if (ipif != NULL) {
10227                                 ipif_get_name(ipif,
10228                                     ipm.ipGroupMemberIfIndex.o_bytes,
10229                                     OCTET_LENGTH);
10230                         } else {
10231                                 ill_get_name(ill,
10232                                     ipm.ipGroupMemberIfIndex.o_bytes,
10233                                     OCTET_LENGTH);
10234                         }
10235                         ipm.ipGroupMemberIfIndex.o_length =
10236                             mi_strlen(ipm.ipGroupMemberIfIndex.o_bytes);

10238                         ipm.ipGroupMemberAddress = ilm->ilm_addr;
10239                         ipm.ipGroupMemberRefCnt = ilm->ilm_refcnt;
10240                         ipm.ipGroupMemberFilterMode = ilm->ilm_fmode;
10241                         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10242                             (char *)&ipm, (int)sizeof (ipm))) {
10243                             ip1dbg(("ip_snmp_get_mib2_ip_group: "
10244                                 "failed to allocate %u bytes\n",
10245                                 (uint_t)sizeof (ipm)));
10246                         }
10247                 }
10248                 rw_exit(&ill->ill_mcast_lock);
10249                 ill_refrele(ill);
10250                 rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10251         }
10252         rw_exit(&ipst->ips_ill_g_lock);
10253         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10254         ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10255             (int)optp->level, (int)optp->name, (int)optp->len));
10256         qreply(q, mpctl);
10257         return (mp2ctl);
10258 }

10260 /* IPv6 multicast group membership. */
10261 static mblk_t *
10262 ip_snmp_get_mib2_ip6_group_mem(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10263 {
10264         struct opthdr           *optp;
10265         mblk_t                  *mp2ctl;
10266         ill_t                   *ill;
10267         ilm_t                   *ilm;
10268         ipv6_member_t           ipm6;
10269         mblk_t                  *mp_tail = NULL;
10270         ill_walk_context_t      ctx;
10271         zoneid_t                zoneid;

10273         /*
10274          * make a copy of the original message
10275          */
10276         mp2ctl = copymsg(mpctl);
10277         zoneid = Q_TO_CONN(q)->conn_zoneid;

10279         /* ip6GroupMember table */
10280         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10281         optp->level = MIB2_IP6;
10282         optp->name = EXPER_IP6_GROUP_MEMBERSHIP;

10284         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10285         ill = ILL_START_WALK_V6(&ctx, ipst);
10286         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10287                 /* Make sure the ill isn't going away. */
10288                 if (!ill_check_and_refhold(ill))
10289                         continue;
10290                 rw_exit(&ipst->ips_ill_g_lock);
10291                 /*
```

```
10292                  * Normally we don't have any members on under IPMP interfaces.
10293                  * We report them as a debugging aid.
10294                  */
10295                 rw_enter(&ill->ill_mcast_lock, RW_READER);
10296                 ipm6.ipv6GroupMemberIfIndex = ill->ill_phyint->phyint_ifindex;
10297                 for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10298                         if (ilm->ilm_zoneid != zoneid &&
10299                             ilm->ilm_zoneid != ALL_ZONES)
10300                                 continue;       /* not this zone */
10301                         ipm6.ipv6GroupMemberAddress = ilm->ilm_v6addr;
10302                         ipm6.ipv6GroupMemberRefCnt = ilm->ilm_refcnt;
10303                         ipm6.ipv6GroupMemberFilterMode = ilm->ilm_fmode;
10304                         if (!snmp_append_data2(mpctl->b_cont,
10305                             &mp_tail,
10306                             (char *)&ipm6, (int)sizeof (ipm6))) {
10307                                 ip1dbg(("ip_snmp_get_mib2_ip6_group: "
10308                                     "failed to allocate %u bytes\n",
10309                                     (uint_t)sizeof (ipm6)));
10310                         }
10311                 }
10312                 rw_exit(&ill->ill_mcast_lock);
10313                 ill_refrele(ill);
10314                 rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10315         }
10316         rw_exit(&ipst->ips_ill_g_lock);

10318         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10319         ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10320             (int)optp->level, (int)optp->name, (int)optp->len));
10321         qreply(q, mpctl);
10322         return (mp2ctl);
10323 }

10325 /* IP multicast filtered sources */
10326 static mblk_t *
10327 ip_snmp_get_mib2_ip_group_src(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10328 {
10329         struct opthdr           *optp;
10330         mblk_t                  *mp2ctl;
10331         ill_t                   *ill;
10332         ipif_t                  *ipif;
10333         ilm_t                   *ilm;
10334         ip_grpsrc_t             ips;
10335         mblk_t                  *mp_tail = NULL;
10336         ill_walk_context_t      ctx;
10337         zoneid_t                zoneid;
10338         int                     i;
10339         slist_t                 *sl;

10341         /*
10342          * make a copy of the original message
10343          */
10344         mp2ctl = copymsg(mpctl);
10345         zoneid = Q_TO_CONN(q)->conn_zoneid;

10347         /* ipGroupSource table */
10348         optp = (struct opthdr *)&mpctl->b_rptr[
10349             sizeof (struct T_optmgmt_ack)];
10350         optp->level = MIB2_IP;
10351         optp->name = EXPER_IP_GROUP_SOURCES;

10353         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10354         ill = ILL_START_WALK_V4(&ctx, ipst);
10355         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10356                 /* Make sure the ill isn't going away. */
10357                 if (!ill_check_and_refhold(ill))
```



```
10358                         continue;
10359                 rw_exit(&ipst->ips_ill_g_lock);
10360                 rw_enter(&ill->ill_mcast_lock, RW_READER);
10361                 for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10362                         sl = ilm->ilm_filter;
10363                         if (ilm->ilm_zoneid != zoneid &&
10364                             ilm->ilm_zoneid != ALL_ZONES)
10365                                 continue;
10366                         if (SLIST_IS_EMPTY(sl))
10367                                 continue;

10369                         /* Is there an ipif for ilm_ifaddr? */
10370                         for (ipif = ill->ill_ipif; ipif != NULL;
10371                             ipif = ipif->ipif_next) {
10372                                 if (!IPIF_IS_CONDEMNED(ipif) &&
10373                                     ipif->ipif_lcl_addr == ilm->ilm_ifaddr &&
10374                                     ilm->ilm_ifaddr != INADDR_ANY)
10375                                         break;
10376                         }
10377                         if (ipif != NULL) {
10378                                 ipif_get_name(ipif,
10379                                     ips.ipGroupSourceIfIndex.o_bytes,
10380                                     OCTET_LENGTH);
10381                         } else {
10382                                 ill_get_name(ill,
10383                                     ips.ipGroupSourceIfIndex.o_bytes,
10384                                     OCTET_LENGTH);
10385                         }
10386                         ips.ipGroupSourceIfIndex.o_length =
10387                             mi_strlen(ips.ipGroupSourceIfIndex.o_bytes);

10389                         ips.ipGroupSourceGroup = ilm->ilm_addr;
10390                         for (i = 0; i < sl->sl_numsrc; i++) {
10391                                 if (!IN6_IS_ADDR_V4MAPPED(&sl->sl_addr[i]))
10392                                         continue;
10393                                 IN6_V4MAPPED_TO_IPADDR(&sl->sl_addr[i],
10394                                     ips.ipGroupSourceAddress);
10395                                 if (snmp_append_data2(mpctl->b_cont, &mp_tail,
10396                                     (char *)&ips, (int)sizeof (ips)) == 0) {
10397                                         ip1dbg(("ip_snmp_get_mib2_ip_group_src:"
10398                                             " failed to allocate %u bytes\n",
10399                                             (uint_t)sizeof (ips)));
10400                                 }
10401                         }
10402                 }
10403                 rw_exit(&ill->ill_mcast_lock);
10404                 ill_refrele(ill);
10405                 rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10406         }
10407         rw_exit(&ipst->ips_ill_g_lock);
10408         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10409         ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10410             (int)optp->level, (int)optp->name, (int)optp->len));
10411         qreply(q, mpctl);
10412         return (mp2ctl);
10413 }

10415 /* IPv6 multicast filtered sources. */
10416 static mblk_t *
10417 ip_snmp_get_mib2_ip6_group_src(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10418 {
10419         struct opthdr           *optp;
10420         mblk_t                  *mp2ctl;
10421         ill_t                   *ill;
10422         ilm_t                   *ilm;
10423         ipv6_grpsrc_t           ips6;
```

```
10424         mblk_t                  *mp_tail = NULL;
10425         ill_walk_context_t      ctx;
10426         zoneid_t                zoneid;
10427         int                     i;
10428         slist_t                 *sl;

10430         /*
10431          * make a copy of the original message
10432          */
10433         mp2ctl = copymsg(mpctl);
10434         zoneid = Q_TO_CONN(q)->conn_zoneid;

10436         /* ip6GroupMember table */
10437         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10438         optp->level = MIB2_IP6;
10439         optp->name = EXPER_IP6_GROUP_SOURCES;

10441         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10442         ill = ILL_START_WALK_V6(&ctx, ipst);
10443         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10444                 /* Make sure the ill isn't going away. */
10445                 if (!ill_check_and_refhold(ill))
10446                         continue;
10447                 rw_exit(&ipst->ips_ill_g_lock);
10448                 /*
10449                  * Normally we don't have any members on under IPMP interfaces.
10450                  * We report them as a debugging aid.
10451                  */
10452                 rw_enter(&ill->ill_mcast_lock, RW_READER);
10453                 ips6.ipv6GroupSourceIfIndex = ill->ill_phyint->phyint_ifindex;
10454                 for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10455                         sl = ilm->ilm_filter;
10456                         if (ilm->ilm_zoneid != zoneid &&
10457                             ilm->ilm_zoneid != ALL_ZONES)
10458                                 continue;
10459                         if (SLIST_IS_EMPTY(sl))
10460                                 continue;
10461                         ips6.ipv6GroupSourceGroup = ilm->ilm_v6addr;
10462                         for (i = 0; i < sl->sl_numsrc; i++) {
10463                                 ips6.ipv6GroupSourceAddress = sl->sl_addr[i];
10464                                 if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10465                                     (char *)&ips6, (int)sizeof (ips6))) {
10466                                         ip1dbg(("ip_snmp_get_mib2_ip6_"
10467                                             "group_src: failed to allocate "
10468                                             "%u bytes\n",
10469                                             (uint_t)sizeof (ips6)));
10470                                 }
10471                         }
10472                 }
10473                 rw_exit(&ill->ill_mcast_lock);
10474                 ill_refrele(ill);
10475                 rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10476         }
10477         rw_exit(&ipst->ips_ill_g_lock);

10479         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10480         ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10481             (int)optp->level, (int)optp->name, (int)optp->len));
10482         qreply(q, mpctl);
10483         return (mp2ctl);
10484 }

10486 /* Multicast routing virtual interface table. */
10487 static mblk_t *
10488 ip_snmp_get_mib2_virt_multi(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10489 {
```



```
10490         struct opthdr           *optp;
10491         mblk_t                  *mp2ctl;

10493         /*
10494          * make a copy of the original message
10495          */
10496         mp2ctl = copymsg(mpctl);

10498         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10499         optp->level = EXPER_DVMRP;
10500         optp->name = EXPER_DVMRP_VIF;
10501         if (!ip_mroute_vif(mpctl->b_cont, ipst)) {
10502                 ip0dbg(("ip_mroute_vif: failed\n"));
10503         }
10504         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10505         ip3dbg(("ip_snmp_get_mib2_virt_multi: level %d, name %d, len %d\n",
10506             (int)optp->level, (int)optp->name, (int)optp->len));
10507         qreply(q, mpctl);
10508         return (mp2ctl);
10509 }

10511 /* Multicast routing table. */
10512 static mblk_t *
10513 ip_snmp_get_mib2_multi_rtable(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10514 {
10515         struct opthdr           *optp;
10516         mblk_t                  *mp2ctl;

10518         /*
10519          * make a copy of the original message
10520          */
10521         mp2ctl = copymsg(mpctl);

10523         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10524         optp->level = EXPER_DVMRP;
10525         optp->name = EXPER_DVMRP_MRT;
10526         if (!ip_mroute_mrt(mpctl->b_cont, ipst)) {
10527                 ip0dbg(("ip_mroute_mrt: failed\n"));
10528         }
10529         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10530         ip3dbg(("ip_snmp_get_mib2_multi_rtable: level %d, name %d, len %d\n",
10531             (int)optp->level, (int)optp->name, (int)optp->len));
10532         qreply(q, mpctl);
10533         return (mp2ctl);
10534 }

10536 /*
10537  * Return ipRouteEntryTable, ipNetToMediaEntryTable, and ipRouteAttributeTable
10538  * in one IRE walk.
10539  */
10540 static mblk_t *
10541 ip_snmp_get_mib2_ip_route_media(queue_t *q, mblk_t *mpctl, int level,
10542     ip_stack_t *ipst)
10543 {
10544         struct opthdr   *optp;
10545         mblk_t          *mp2ctl;        /* Returned */
10546         mblk_t          *mp3ctl;        /* nettomedia */
10547         mblk_t          *mp4ctl;        /* routeattrs */
10548         iproutedata_t   ird;
10549         zoneid_t        zoneid;

10551         /*
10552          * make copies of the original message
10553          *      - mp2ctl is returned unchanged to the caller for his use
10554          *      - mpctl is sent upstream as ipRouteEntryTable
10555          *      - mp3ctl is sent upstream as ipNetToMediaEntryTable
```

```
10556          *      - mp4ctl is sent upstream as ipRouteAttributeTable
10557          */
10558         mp2ctl = copymsg(mpctl);
10559         mp3ctl = copymsg(mpctl);
10560         mp4ctl = copymsg(mpctl);
10561         if (mp3ctl == NULL || mp4ctl == NULL) {
10562                 freemsg(mp4ctl);
10563                 freemsg(mp3ctl);
10564                 freemsg(mp2ctl);
10565                 freemsg(mpctl);
10566                 return (NULL);
10567         }

10569         bzero(&ird, sizeof (ird));

10571         ird.ird_route.lp_head = mpctl->b_cont;
10572         ird.ird_netmedia.lp_head = mp3ctl->b_cont;
10573         ird.ird_attrs.lp_head = mp4ctl->b_cont;
10574         /*
10575          * If the level has been set the special EXPER_IP_AND_ALL_IRES value,
10576          * then also include ire_testhidden IREs and IRE_IF_CLONE.  This is
10577          * intended a temporary solution until a proper MIB API is provided
10578          * that provides complete filtering/caller-opt-in.
10579          */
10580         if (level == EXPER_IP_AND_ALL_IRES)
10581                 ird.ird_flags |= IRD_REPORT_ALL;

10583         zoneid = Q_TO_CONN(q)->conn_zoneid;
10584         ire_walk_v4(ip_snmp_get2_v4, &ird, zoneid, ipst);

10586         /* ipRouteEntryTable in mpctl */
10587         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10588         optp->level = MIB2_IP;
10589         optp->name = MIB2_IP_ROUTE;
10590         optp->len = msgdsize(ird.ird_route.lp_head);
10591         ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10592             (int)optp->level, (int)optp->name, (int)optp->len));
10593         qreply(q, mpctl);

10595         /* ipNetToMediaEntryTable in mp3ctl */
10596         ncec_walk(NULL, ip_snmp_get2_v4_media, &ird, ipst);

10598         optp = (struct opthdr *)&mp3ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10599         optp->level = MIB2_IP;
10600         optp->name = MIB2_IP_MEDIA;
10601         optp->len = msgdsize(ird.ird_netmedia.lp_head);
10602         ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10603             (int)optp->level, (int)optp->name, (int)optp->len));
10604         qreply(q, mp3ctl);

10606         /* ipRouteAttributeTable in mp4ctl */
10607         optp = (struct opthdr *)&mp4ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10608         optp->level = MIB2_IP;
10609         optp->name = EXPER_IP_RTATTR;
10610         optp->len = msgdsize(ird.ird_attrs.lp_head);
10611         ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10612             (int)optp->level, (int)optp->name, (int)optp->len));
10613         if (optp->len == 0)
10614                 freemsg(mp4ctl);
10615         else
10616                 qreply(q, mp4ctl);

10618         return (mp2ctl);
10619 }

10621 /*
```



```
10622  * Return ipv6RouteEntryTable and ipv6RouteAttributeTable in one IRE walk, and
10623  * ipv6NetToMediaEntryTable in an NDP walk.
10624  */
10625 static mblk_t *
10626 ip_snmp_get_mib2_ip6_route_media(queue_t *q, mblk_t *mpctl, int level,
10627     ip_stack_t *ipst)
10628 {
10629         struct opthdr   *optp;
10630         mblk_t          *mp2ctl;        /* Returned */
10631         mblk_t          *mp3ctl;        /* nettomedia */
10632         mblk_t          *mp4ctl;        /* routeattrs */
10633         iproutedata_t   ird;
10634         zoneid_t        zoneid;

10636         /*
10637          * make copies of the original message
10638          *      - mp2ctl is returned unchanged to the caller for his use
10639          *      - mpctl is sent upstream as ipv6RouteEntryTable
10640          *      - mp3ctl is sent upstream as ipv6NetToMediaEntryTable
10641          *      - mp4ctl is sent upstream as ipv6RouteAttributeTable
10642          */
10643         mp2ctl = copymsg(mpctl);
10644         mp3ctl = copymsg(mpctl);
10645         mp4ctl = copymsg(mpctl);
10646         if (mp3ctl == NULL || mp4ctl == NULL) {
10647                 freemsg(mp4ctl);
10648                 freemsg(mp3ctl);
10649                 freemsg(mp2ctl);
10650                 freemsg(mpctl);
10651                 return (NULL);
10652         }

10654         bzero(&ird, sizeof (ird));

10656         ird.ird_route.lp_head = mpctl->b_cont;
10657         ird.ird_netmedia.lp_head = mp3ctl->b_cont;
10658         ird.ird_attrs.lp_head = mp4ctl->b_cont;
10659         /*
10660          * If the level has been set the special EXPER_IP_AND_ALL_IRES value,
10661          * then also include ire_testhidden IREs and IRE_IF_CLONE.  This is
10662          * intended a temporary solution until a proper MIB API is provided
10663          * that provides complete filtering/caller-opt-in.
10664          */
10665         if (level == EXPER_IP_AND_ALL_IRES)
10666                 ird.ird_flags |= IRD_REPORT_ALL;

10668         zoneid = Q_TO_CONN(q)->conn_zoneid;
10669         ire_walk_v6(ip_snmp_get2_v6_route, &ird, zoneid, ipst);

10671         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10672         optp->level = MIB2_IP6;
10673         optp->name = MIB2_IP6_ROUTE;
10674         optp->len = msgdsize(ird.ird_route.lp_head);
10675         ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10676             (int)optp->level, (int)optp->name, (int)optp->len));
10677         qreply(q, mpctl);

10679         /* ipv6NetToMediaEntryTable in mp3ctl */
10680         ncec_walk(NULL, ip_snmp_get2_v6_media, &ird, ipst);

10682         optp = (struct opthdr *)&mp3ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10683         optp->level = MIB2_IP6;
10684         optp->name = MIB2_IP6_MEDIA;
10685         optp->len = msgdsize(ird.ird_netmedia.lp_head);
10686         ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10687             (int)optp->level, (int)optp->name, (int)optp->len));
```

```
10688         qreply(q, mp3ctl);

10690         /* ipv6RouteAttributeTable in mp4ctl */
10691         optp = (struct opthdr *)&mp4ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10692         optp->level = MIB2_IP6;
10693         optp->name = EXPER_IP_RTATTR;
10694         optp->len = msgdsize(ird.ird_attrs.lp_head);
10695         ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10696             (int)optp->level, (int)optp->name, (int)optp->len));
10697         if (optp->len == 0)
10698                 freemsg(mp4ctl);
10699         else
10700                 qreply(q, mp4ctl);

10702         return (mp2ctl);
10703 }

10705 /*
10706  * IPv6 mib: One per ill
10707  */
10708 static mblk_t *
10709 ip_snmp_get_mib2_ip6(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
10710     boolean_t legacy_req)
10711 {
10712         struct opthdr           *optp;
10713         mblk_t                  *mp2ctl;
10714         ill_t                   *ill;
10715         ill_walk_context_t      ctx;
10716         mblk_t                  *mp_tail = NULL;
10717         mib2_ipv6AddrEntry_t    mae6;
10718         mib2_ipIfStatsEntry_t   *ise;
10719         size_t                  ise_size, iae_size;

10721         /*
10722          * Make a copy of the original message
10723          */
10724         mp2ctl = copymsg(mpctl);

10726         /* fixed length IPv6 structure ... */

10728         if (legacy_req) {
10729                 ise_size = LEGACY_MIB_SIZE(&ipst->ips_ip6_mib,
10730                     mib2_ipIfStatsEntry_t);
10731                 iae_size = LEGACY_MIB_SIZE(&mae6, mib2_ipv6AddrEntry_t);
10732         } else {
10733                 ise_size = sizeof (mib2_ipIfStatsEntry_t);
10734                 iae_size = sizeof (mib2_ipv6AddrEntry_t);
10735         }

10737         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10738         optp->level = MIB2_IP6;
10739         optp->name = 0;
10740         /* Include "unknown interface" ip6_mib */
10741         ipst->ips_ip6_mib.ipIfStatsIPVersion = MIB2_INETADDRESSTYPE_ipv6;
10742         ipst->ips_ip6_mib.ipIfStatsIfIndex =
10743             MIB2_UNKNOWN_INTERFACE; /* Flag to netstat */
10744         SET_MIB(ipst->ips_ip6_mib.ipIfStatsForwarding,
10745             ipst->ips_ipv6_forwarding ? 1 : 2);
10746         SET_MIB(ipst->ips_ip6_mib.ipIfStatsDefaultHopLimit,
10747             ipst->ips_ipv6_def_hops);
10748         SET_MIB(ipst->ips_ip6_mib.ipIfStatsEntrySize,
10749             sizeof (mib2_ipIfStatsEntry_t));
10750         SET_MIB(ipst->ips_ip6_mib.ipIfStatsAddrEntrySize,
10751             sizeof (mib2_ipv6AddrEntry_t));
10752         SET_MIB(ipst->ips_ip6_mib.ipIfStatsRouteEntrySize,
10753             sizeof (mib2_ipv6RouteEntry_t));
```



```
10754         SET_MIB(ipst->ips_ip6_mib.ipIfStatsNetToMediaEntrySize,
10755             sizeof (mib2_ipv6NetToMediaEntry_t));
10756         SET_MIB(ipst->ips_ip6_mib.ipIfStatsMemberEntrySize,
10757             sizeof (ipv6_member_t));
10758         SET_MIB(ipst->ips_ip6_mib.ipIfStatsGroupSourceEntrySize,
10759             sizeof (ipv6_grpsrc_t));

10761         /*
10762          * Synchronize 64- and 32-bit counters
10763          */
10764         SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInReceives,
10765             ipIfStatsHCInReceives);
10766         SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInDelivers,
10767             ipIfStatsHCInDelivers);
10768         SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutRequests,
10769             ipIfStatsHCOutRequests);
10770         SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutForwDatagrams,
10771             ipIfStatsHCOutForwDatagrams);
10772         SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutMcastPkts,
10773             ipIfStatsHCOutMcastPkts);
10774         SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInMcastPkts,
10775             ipIfStatsHCInMcastPkts);

10777         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10778             (char *)&ipst->ips_ip6_mib, (int)ise_size)) {
10779                 ip1dbg(("ip_snmp_get_mib2_ip6: failed to allocate %u bytes\n",
10780                     (uint_t)ise_size));
10781         } else if (legacy_req) {
10782                 /* Adjust the EntrySize fields for legacy requests. */
10783                 ise =
10784                     (mib2_ipIfStatsEntry_t *)(mp_tail->b_wptr - (int)ise_size);
10785                 SET_MIB(ise->ipIfStatsEntrySize, ise_size);
10786                 SET_MIB(ise->ipIfStatsAddrEntrySize, iae_size);
10787         }

10789         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10790         ill = ILL_START_WALK_V6(&ctx, ipst);
10791         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10792                 ill->ill_ip_mib->ipIfStatsIfIndex =
10793                     ill->ill_phyint->phyint_ifindex;
10794                 SET_MIB(ill->ill_ip_mib->ipIfStatsForwarding,
10795                     ipst->ips_ipv6_forwarding ? 1 : 2);
10796                 SET_MIB(ill->ill_ip_mib->ipIfStatsDefaultHopLimit,
10797                     ill->ill_max_hops);

10799                 /*
10800                  * Synchronize 64- and 32-bit counters
10801                  */
10802                 SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInReceives,
10803                     ipIfStatsHCInReceives);
10804                 SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInDelivers,
10805                     ipIfStatsHCInDelivers);
10806                 SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutRequests,
10807                     ipIfStatsHCOutRequests);
10808                 SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutForwDatagrams,
10809                     ipIfStatsHCOutForwDatagrams);
10810                 SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutMcastPkts,
10811                     ipIfStatsHCOutMcastPkts);
10812                 SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInMcastPkts,
10813                     ipIfStatsHCInMcastPkts);

10815                 if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10816                     (char *)ill->ill_ip_mib, (int)ise_size)) {
10817                         ip1dbg(("ip_snmp_get_mib2_ip6: failed to allocate "
10818                         "%u bytes\n", (uint_t)ise_size));
10819                 } else if (legacy_req) {
```

```
10820                         /* Adjust the EntrySize fields for legacy requests. */
10821                         ise = (mib2_ipIfStatsEntry_t *)(mp_tail->b_wptr -
10822                             (int)ise_size);
10823                         SET_MIB(ise->ipIfStatsEntrySize, ise_size);
10824                         SET_MIB(ise->ipIfStatsAddrEntrySize, iae_size);
10825                 }
10826         }
10827         rw_exit(&ipst->ips_ill_g_lock);

10829         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10830         ip3dbg(("ip_snmp_get_mib2_ip6: level %d, name %d, len %d\n",
10831             (int)optp->level, (int)optp->name, (int)optp->len));
10832         qreply(q, mpctl);
10833         return (mp2ctl);
10834 }

10836 /*
10837  * ICMPv6 mib: One per ill
10838  */
10839 static mblk_t *
10840 ip_snmp_get_mib2_icmp6(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10841 {
10842         struct opthdr           *optp;
10843         mblk_t                  *mp2ctl;
10844         ill_t                   *ill;
10845         ill_walk_context_t      ctx;
10846         mblk_t                  *mp_tail = NULL;
10847         /*
10848          * Make a copy of the original message
10849          */
10850         mp2ctl = copymsg(mpctl);

10852         /* fixed length ICMPv6 structure ... */

10854         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10855         optp->level = MIB2_ICMP6;
10856         optp->name = 0;
10857         /* Include "unknown interface" icmp6_mib */
10858         ipst->ips_icmp6_mib.ipv6IfIcmpIfIndex =
10859             MIB2_UNKNOWN_INTERFACE; /* netstat flag */
10860         ipst->ips_icmp6_mib.ipv6IfIcmpEntrySize =
10861             sizeof (mib2_ipv6IfIcmpEntry_t);
10862         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10863             (char *)&ipst->ips_icmp6_mib,
10864             (int)sizeof (ipst->ips_icmp6_mib))) {
10865                 ip1dbg(("ip_snmp_get_mib2_icmp6: failed to allocate %u bytes\n",
10866                     (uint_t)sizeof (ipst->ips_icmp6_mib)));
10867         }

10869         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10870         ill = ILL_START_WALK_V6(&ctx, ipst);
10871         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10872                 ill->ill_icmp6_mib->ipv6IfIcmpIfIndex =
10873                     ill->ill_phyint->phyint_ifindex;
10874                 if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10875                     (char *)ill->ill_icmp6_mib,
10876                     (int)sizeof (*ill->ill_icmp6_mib))) {
10877                         ip1dbg(("ip_snmp_get_mib2_icmp6: failed to allocate "
10878                             "%u bytes\n",
10879                             (uint_t)sizeof (*ill->ill_icmp6_mib)));
10880                 }
10881         }
10882         rw_exit(&ipst->ips_ill_g_lock);

10884         optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10885         ip3dbg(("ip_snmp_get_mib2_icmp6: level %d, name %d, len %d\n",
```



```
10886             (int)optp->level, (int)optp->name, (int)optp->len));
10887         qreply(q, mpctl);
10888         return (mp2ctl);
10889 }

10891 /*
10892  * ire_walk routine to create both ipRouteEntryTable and
10893  * ipRouteAttributeTable in one IRE walk
10894  */
10895 static void
10896 ip_snmp_get2_v4(ire_t *ire, iproutedata_t *ird)
10897 {
10898         ill_t                           *ill;
10899         mib2_ipRouteEntry_t             *re;
10900         mib2_ipAttributeEntry_t         iaes;
10901         tsol_ire_gw_secattr_t           *attrp;
10902         tsol_gc_t                       *gc = NULL;
10903         tsol_gcgrp_t                    *gcgrp = NULL;
10904         ip_stack_t                      *ipst = ire->ire_ipst;

10906         ASSERT(ire->ire_ipversion == IPV4_VERSION);

10908         if (!(ird->ird_flags & IRD_REPORT_ALL)) {
10909                 if (ire->ire_testhidden)
10910                         return;
10911                 if (ire->ire_type & IRE_IF_CLONE)
10912                         return;
10913         }

10915         if ((re = kmem_zalloc(sizeof (*re), KM_NOSLEEP)) == NULL)
10916                 return;

10918         if ((attrp = ire->ire_gw_secattr) != NULL) {
10919                 mutex_enter(&attrp->igsa_lock);
10920                 if ((gc = attrp->igsa_gc) != NULL) {
10921                         gcgrp = gc->gc_grp;
10922                         ASSERT(gcgrp != NULL);
10923                         rw_enter(&gcgrp->gcgrp_rwlock, RW_READER);
10924                 }
10925                 mutex_exit(&attrp->igsa_lock);
10926         }
10927         /*
10928          * Return all IRE types for route table... let caller pick and choose
10929          */
10930         re->ipRouteDest = ire->ire_addr;
10931         ill = ire->ire_ill;
10932         re->ipRouteIfIndex.o_length = 0;
10933         if (ill != NULL) {
10934                 ill_get_name(ill, re->ipRouteIfIndex.o_bytes, OCTET_LENGTH);
10935                 re->ipRouteIfIndex.o_length =
10936                     mi_strlen(re->ipRouteIfIndex.o_bytes);
10937         }
10938         re->ipRouteMetric1 = -1;
10939         re->ipRouteMetric2 = -1;
10940         re->ipRouteMetric3 = -1;
10941         re->ipRouteMetric4 = -1;

10943         re->ipRouteNextHop = ire->ire_gateway_addr;
10944         /* indirect(4), direct(3), or invalid(2) */
10945         if (ire->ire_flags & (RTF_REJECT | RTF_BLACKHOLE))
10946                 re->ipRouteType = 2;
10947         else if (ire->ire_type & IRE_ONLINK)
10948                 re->ipRouteType = 3;
10949         else
10950                 re->ipRouteType = 4;
```

```
10952         re->ipRouteProto = -1;
10953         re->ipRouteAge = gethrestime_sec() - ire->ire_create_time;
10954         re->ipRouteMask = ire->ire_mask;
10955         re->ipRouteMetric5 = -1;
10956         re->ipRouteInfo.re_max_frag = ire->ire_metrics.iulp_mtu;
10957         if (ire->ire_ill != NULL && re->ipRouteInfo.re_max_frag == 0)
10958                 re->ipRouteInfo.re_max_frag = ire->ire_ill->ill_mtu;

10960         re->ipRouteInfo.re_frag_flag    = 0;
10961         re->ipRouteInfo.re_rtt          = 0;
10962         re->ipRouteInfo.re_src_addr     = 0;
10963         re->ipRouteInfo.re_ref          = ire->ire_refcnt;
10964         re->ipRouteInfo.re_obpkt        = ire->ire_ob_pkt_count;
10965         re->ipRouteInfo.re_ibpkt        = ire->ire_ib_pkt_count;
10966         re->ipRouteInfo.re_flags        = ire->ire_flags;

10968         /* Add the IRE_IF_CLONE's counters to their parent IRE_INTERFACE */
10969         if (ire->ire_type & IRE_INTERFACE) {
10970                 ire_t *child;

10972                 rw_enter(&ipst->ips_ire_dep_lock, RW_READER);
10973                 child = ire->ire_dep_children;
10974                 while (child != NULL) {
10975                         re->ipRouteInfo.re_obpkt += child->ire_ob_pkt_count;
10976                         re->ipRouteInfo.re_ibpkt += child->ire_ib_pkt_count;
10977                         child = child->ire_dep_sib_next;
10978                 }
10979                 rw_exit(&ipst->ips_ire_dep_lock);
10980         }

10982         if (ire->ire_flags & RTF_DYNAMIC) {
10983                 re->ipRouteInfo.re_ire_type     = IRE_HOST_REDIRECT;
10984         } else {
10985                 re->ipRouteInfo.re_ire_type     = ire->ire_type;
10986         }

10988         if (!snmp_append_data2(ird->ird_route.lp_head, &ird->ird_route.lp_tail,
10989             (char *)re, (int)sizeof (*re))) {
10990                 ip1dbg(("ip_snmp_get2_v4: failed to allocate %u bytes\n",
10991                     (uint_t)sizeof (*re)));
10992         }

10994         if (gc != NULL) {
10995                 iaes.iae_routeidx = ird->ird_idx;
10996                 iaes.iae_doi = gc->gc_db->gcdb_doi;
10997                 iaes.iae_slrange = gc->gc_db->gcdb_slrange;

10999                 if (!snmp_append_data2(ird->ird_attrs.lp_head,
11000                     &ird->ird_attrs.lp_tail, (char *)&iaes, sizeof (iaes))) {
11001                         ip1dbg(("ip_snmp_get2_v4: failed to allocate %u "
11002                             "bytes\n", (uint_t)sizeof (iaes)));
11003                 }
11004         }

11006         /* bump route index for next pass */
11007         ird->ird_idx++;

11009         kmem_free(re, sizeof (*re));
11010         if (gcgrp != NULL)
11011                 rw_exit(&gcgrp->gcgrp_rwlock);
11012 }

11014 /*
11015  * ire_walk routine to create ipv6RouteEntryTable and ipRouteEntryTable.
11016  */
11017 static void
```



```
11018 ip_snmp_get2_v6_route(ire_t *ire, iproutedata_t *ird)
11019 {
11020         ill_t                           *ill;
11021         mib2_ipv6RouteEntry_t           *re;
11022         mib2_ipAttributeEntry_t         iaes;
11023         tsol_ire_gw_secattr_t           *attrp;
11024         tsol_gc_t                       *gc = NULL;
11025         tsol_gcgrp_t                    *gcgrp = NULL;
11026         ip_stack_t                      *ipst = ire->ire_ipst;

11028         ASSERT(ire->ire_ipversion == IPV6_VERSION);

11030         if (!(ird->ird_flags & IRD_REPORT_ALL)) {
11031                 if (ire->ire_testhidden)
11032                         return;
11033                 if (ire->ire_type & IRE_IF_CLONE)
11034                         return;
11035         }

11037         if ((re = kmem_zalloc(sizeof (*re), KM_NOSLEEP)) == NULL)
11038                 return;

11040         if ((attrp = ire->ire_gw_secattr) != NULL) {
11041                 mutex_enter(&attrp->igsa_lock);
11042                 if ((gc = attrp->igsa_gc) != NULL) {
11043                         gcgrp = gc->gc_grp;
11044                         ASSERT(gcgrp != NULL);
11045                         rw_enter(&gcgrp->gcgrp_rwlock, RW_READER);
11046                 }
11047                 mutex_exit(&attrp->igsa_lock);
11048         }
11049         /*
11050          * Return all IRE types for route table... let caller pick and choose
11051          */
11052         re->ipv6RouteDest = ire->ire_addr_v6;
11053         re->ipv6RoutePfxLength = ip_mask_to_plen_v6(&ire->ire_mask_v6);
11054         re->ipv6RouteIndex = 0; /* Unique when multiple with same dest/plen */
11055         re->ipv6RouteIfIndex.o_length = 0;
11056         ill = ire->ire_ill;
11057         if (ill != NULL) {
11058                 ill_get_name(ill, re->ipv6RouteIfIndex.o_bytes, OCTET_LENGTH);
11059                 re->ipv6RouteIfIndex.o_length =
11060                     mi_strlen(re->ipv6RouteIfIndex.o_bytes);
11061         }

11063         ASSERT(!(ire->ire_type & IRE_BROADCAST));

11065         mutex_enter(&ire->ire_lock);
11066         re->ipv6RouteNextHop = ire->ire_gateway_addr_v6;
11067         mutex_exit(&ire->ire_lock);

11069         /* remote(4), local(3), or discard(2) */
11070         if (ire->ire_flags & (RTF_REJECT | RTF_BLACKHOLE))
11071                 re->ipv6RouteType = 2;
11072         else if (ire->ire_type & IRE_ONLINK)
11073                 re->ipv6RouteType = 3;
11074         else
11075                 re->ipv6RouteType = 4;

11077         re->ipv6RouteProtocol   = -1;
11078         re->ipv6RoutePolicy     = 0;
11079         re->ipv6RouteAge        = gethrestime_sec() - ire->ire_create_time;
11080         re->ipv6RouteNextHopRDI = 0;
11081         re->ipv6RouteWeight     = 0;
11082         re->ipv6RouteMetric     = 0;
11083         re->ipv6RouteInfo.re_max_frag = ire->ire_metrics.iulp_mtu;
```

```
11084         if (ire->ire_ill != NULL && re->ipv6RouteInfo.re_max_frag == 0)
11085                 re->ipv6RouteInfo.re_max_frag = ire->ire_ill->ill_mtu;

11087         re->ipv6RouteInfo.re_frag_flag  = 0;
11088         re->ipv6RouteInfo.re_rtt        = 0;
11089         re->ipv6RouteInfo.re_src_addr   = ipv6_all_zeros;
11090         re->ipv6RouteInfo.re_obpkt      = ire->ire_ob_pkt_count;
11091         re->ipv6RouteInfo.re_ibpkt      = ire->ire_ib_pkt_count;
11092         re->ipv6RouteInfo.re_ref        = ire->ire_refcnt;
11093         re->ipv6RouteInfo.re_flags      = ire->ire_flags;

11095         /* Add the IRE_IF_CLONE's counters to their parent IRE_INTERFACE */
11096         if (ire->ire_type & IRE_INTERFACE) {
11097                 ire_t *child;

11099                 rw_enter(&ipst->ips_ire_dep_lock, RW_READER);
11100                 child = ire->ire_dep_children;
11101                 while (child != NULL) {
11102                         re->ipv6RouteInfo.re_obpkt += child->ire_ob_pkt_count;
11103                         re->ipv6RouteInfo.re_ibpkt += child->ire_ib_pkt_count;
11104                         child = child->ire_dep_sib_next;
11105                 }
11106                 rw_exit(&ipst->ips_ire_dep_lock);
11107         }
11108         if (ire->ire_flags & RTF_DYNAMIC) {
11109                 re->ipv6RouteInfo.re_ire_type   = IRE_HOST_REDIRECT;
11110         } else {
11111                 re->ipv6RouteInfo.re_ire_type   = ire->ire_type;
11112         }

11114         if (!snmp_append_data2(ird->ird_route.lp_head, &ird->ird_route.lp_tail,
11115             (char *)re, (int)sizeof (*re))) {
11116                 ip1dbg(("ip_snmp_get2_v6: failed to allocate %u bytes\n",
11117                     (uint_t)sizeof (*re)));
11118         }

11120         if (gc != NULL) {
11121                 iaes.iae_routeidx = ird->ird_idx;
11122                 iaes.iae_doi = gc->gc_db->gcdb_doi;
11123                 iaes.iae_slrange = gc->gc_db->gcdb_slrange;

11125                 if (!snmp_append_data2(ird->ird_attrs.lp_head,
11126                     &ird->ird_attrs.lp_tail, (char *)&iaes, sizeof (iaes))) {
11127                         ip1dbg(("ip_snmp_get2_v6: failed to allocate %u "
11128                             "bytes\n", (uint_t)sizeof (iaes)));
11129                 }
11130         }

11132         /* bump route index for next pass */
11133         ird->ird_idx++;

11135         kmem_free(re, sizeof (*re));
11136         if (gcgrp != NULL)
11137                 rw_exit(&gcgrp->gcgrp_rwlock);
11138 }

11140 /*
11141  * ncec_walk routine to create ipv6NetToMediaEntryTable
11142  */
11143 static int
11144 ip_snmp_get2_v6_media(ncec_t *ncec, iproutedata_t *ird)
11145 {
11146         ill_t                           *ill;
11147         mib2_ipv6NetToMediaEntry_t      ntme;

11149         ill = ncec->ncec_ill;
```



```
11150         /* skip arpce entries, and loopback ncec entries */
11151         if (ill->ill_isv6 == B_FALSE || ill->ill_net_type == IRE_LOOPBACK)
11152                 return (0);
11153         /*
11154          * Neighbor cache entry attached to IRE with on-link
11155          * destination.
11156          * We report all IPMP groups on ncec_ill which is normally the upper.
11157          */
11158         ntme.ipv6NetToMediaIfIndex = ill->ill_phyint->phyint_ifindex;
11159         ntme.ipv6NetToMediaNetAddress = ncec->ncec_addr;
11160         ntme.ipv6NetToMediaPhysAddress.o_length = ill->ill_phys_addr_length;
11161         if (ncec->ncec_lladdr != NULL) {
11162                 bcopy(ncec->ncec_lladdr, ntme.ipv6NetToMediaPhysAddress.o_bytes,
11163                     ntme.ipv6NetToMediaPhysAddress.o_length);
11164         }
11165         /*
11166          * Note: Returns ND_* states. Should be:
11167          * reachable(1), stale(2), delay(3), probe(4),
11168          * invalid(5), unknown(6)
11169          */
11170         ntme.ipv6NetToMediaState = ncec->ncec_state;
11171         ntme.ipv6NetToMediaLastUpdated = 0;

11173         /* other(1), dynamic(2), static(3), local(4) */
11174         if (NCE_MYADDR(ncec)) {
11175                 ntme.ipv6NetToMediaType = 4;
11176         } else if (ncec->ncec_flags & NCE_F_PUBLISH) {
11177                 ntme.ipv6NetToMediaType = 1; /* proxy */
11178         } else if (ncec->ncec_flags & NCE_F_STATIC) {
11179                 ntme.ipv6NetToMediaType = 3;
11180         } else if (ncec->ncec_flags & (NCE_F_MCAST|NCE_F_BCAST)) {
11181                 ntme.ipv6NetToMediaType = 1;
11182         } else {
11183                 ntme.ipv6NetToMediaType = 2;
11184         }

11186         if (!snmp_append_data2(ird->ird_netmedia.lp_head,
11187             &ird->ird_netmedia.lp_tail, (char *)&ntme, sizeof (ntme))) {
11188                 ip1dbg(("ip_snmp_get2_v6_media: failed to allocate %u bytes\n",
11189                     (uint_t)sizeof (ntme)));
11190         }
11191         return (0);
11192 }

11194 int
11195 nce2ace(ncec_t *ncec)
11196 {
11197         int flags = 0;

11199         if (NCE_ISREACHABLE(ncec))
11200                 flags |= ACE_F_RESOLVED;
11201         if (ncec->ncec_flags & NCE_F_AUTHORITY)
11202                 flags |= ACE_F_AUTHORITY;
11203         if (ncec->ncec_flags & NCE_F_PUBLISH)
11204                 flags |= ACE_F_PUBLISH;
11205         if ((ncec->ncec_flags & NCE_F_NONUD) != 0)
11206                 flags |= ACE_F_PERMANENT;
11207         if (NCE_MYADDR(ncec))
11208                 flags |= (ACE_F_MYADDR | ACE_F_AUTHORITY);
11209         if (ncec->ncec_flags & NCE_F_UNVERIFIED)
11210                 flags |= ACE_F_UNVERIFIED;
11211         if (ncec->ncec_flags & NCE_F_AUTHORITY)
11212                 flags |= ACE_F_AUTHORITY;
11213         if (ncec->ncec_flags & NCE_F_DELAYED)
11214                 flags |= ACE_F_DELAYED;
11215         return (flags);
```

```
11216 }

11218 /*
11219  * ncec_walk routine to create ipNetToMediaEntryTable
11220  */
11221 static int
11222 ip_snmp_get2_v4_media(ncec_t *ncec, iproutedata_t *ird)
11223 {
11224         ill_t                           *ill;
11225         mib2_ipNetToMediaEntry_t        ntme;
11226         const char                      *name = "unknown";
11227         ipaddr_t                        ncec_addr;

11229         ill = ncec->ncec_ill;
11230         if (ill->ill_isv6 || (ncec->ncec_flags & NCE_F_BCAST) ||
11231             ill->ill_net_type == IRE_LOOPBACK)
11232                 return (0);

11234         /* We report all IPMP groups on ncec_ill which is normally the upper. */
11235         name = ill->ill_name;
11236         /* Based on RFC 4293: other(1), inval(2), dyn(3), stat(4) */
11237         if (NCE_MYADDR(ncec)) {
11238                 ntme.ipNetToMediaType = 4;
11239         } else if (ncec->ncec_flags & (NCE_F_MCAST|NCE_F_BCAST|NCE_F_PUBLISH)) {
11240                 ntme.ipNetToMediaType = 1;
11241         } else {
11242                 ntme.ipNetToMediaType = 3;
11243         }
11244         ntme.ipNetToMediaIfIndex.o_length = MIN(OCTET_LENGTH, strlen(name));
11245         bcopy(name, ntme.ipNetToMediaIfIndex.o_bytes,
11246             ntme.ipNetToMediaIfIndex.o_length);

11248         IN6_V4MAPPED_TO_IPADDR(&ncec->ncec_addr, ncec_addr);
11249         bcopy(&ncec_addr, &ntme.ipNetToMediaNetAddress, sizeof (ncec_addr));

11251         ntme.ipNetToMediaInfo.ntm_mask.o_length = sizeof (ipaddr_t);
11252         ncec_addr = INADDR_BROADCAST;
11253         bcopy(&ncec_addr, ntme.ipNetToMediaInfo.ntm_mask.o_bytes,
11254             sizeof (ncec_addr));
11255         /*
11256          * map all the flags to the ACE counterpart.
11257          */
11258         ntme.ipNetToMediaInfo.ntm_flags = nce2ace(ncec);

11260         ntme.ipNetToMediaPhysAddress.o_length =
11261             MIN(OCTET_LENGTH, ill->ill_phys_addr_length);

11263         if (!NCE_ISREACHABLE(ncec))
11264                 ntme.ipNetToMediaPhysAddress.o_length = 0;
11265         else {
11266                 if (ncec->ncec_lladdr != NULL) {
11267                         bcopy(ncec->ncec_lladdr,
11268                             ntme.ipNetToMediaPhysAddress.o_bytes,
11269                             ntme.ipNetToMediaPhysAddress.o_length);
11270                 }
11271         }

11273         if (!snmp_append_data2(ird->ird_netmedia.lp_head,
11274             &ird->ird_netmedia.lp_tail, (char *)&ntme, sizeof (ntme))) {
11275                 ip1dbg(("ip_snmp_get2_v4_media: failed to allocate %u bytes\n",
11276                     (uint_t)sizeof (ntme)));
11277         }
11278         return (0);
11279 }

11281 /*
```



```
11282  * return (0) if invalid set request, 1 otherwise, including non-tcp requests
11283  */
11284 /* ARGSUSED */
11285 int
11286 ip_snmp_set(queue_t *q, int level, int name, uchar_t *ptr, int len)
11287 {
11288         switch (level) {
11289         case MIB2_IP:
11290         case MIB2_ICMP:
11291                 switch (name) {
11292                 default:
11293                         break;
11294                 }
11295                 return (1);
11296         default:
11297                 return (1);
11298         }
11299 }

11301 /*
11302  * When there exists both a 64- and 32-bit counter of a particular type
11303  * (i.e., InReceives), only the 64-bit counters are added.
11304  */
11305 void
11306 ip_mib2_add_ip_stats(mib2_ipIfStatsEntry_t *o1, mib2_ipIfStatsEntry_t *o2)
11307 {
11308         UPDATE_MIB(o1, ipIfStatsInHdrErrors, o2->ipIfStatsInHdrErrors);
11309         UPDATE_MIB(o1, ipIfStatsInTooBigErrors, o2->ipIfStatsInTooBigErrors);
11310         UPDATE_MIB(o1, ipIfStatsInNoRoutes, o2->ipIfStatsInNoRoutes);
11311         UPDATE_MIB(o1, ipIfStatsInAddrErrors, o2->ipIfStatsInAddrErrors);
11312         UPDATE_MIB(o1, ipIfStatsInUnknownProtos, o2->ipIfStatsInUnknownProtos);
11313         UPDATE_MIB(o1, ipIfStatsInTruncatedPkts, o2->ipIfStatsInTruncatedPkts);
11314         UPDATE_MIB(o1, ipIfStatsInDiscards, o2->ipIfStatsInDiscards);
11315         UPDATE_MIB(o1, ipIfStatsOutDiscards, o2->ipIfStatsOutDiscards);
11316         UPDATE_MIB(o1, ipIfStatsOutFragOKs, o2->ipIfStatsOutFragOKs);
11317         UPDATE_MIB(o1, ipIfStatsOutFragFails, o2->ipIfStatsOutFragFails);
11318         UPDATE_MIB(o1, ipIfStatsOutFragCreates, o2->ipIfStatsOutFragCreates);
11319         UPDATE_MIB(o1, ipIfStatsReasmReqds, o2->ipIfStatsReasmReqds);
11320         UPDATE_MIB(o1, ipIfStatsReasmOKs, o2->ipIfStatsReasmOKs);
11321         UPDATE_MIB(o1, ipIfStatsReasmFails, o2->ipIfStatsReasmFails);
11322         UPDATE_MIB(o1, ipIfStatsOutNoRoutes, o2->ipIfStatsOutNoRoutes);
11323         UPDATE_MIB(o1, ipIfStatsReasmDuplicates, o2->ipIfStatsReasmDuplicates);
11324         UPDATE_MIB(o1, ipIfStatsReasmPartDups, o2->ipIfStatsReasmPartDups);
11325         UPDATE_MIB(o1, ipIfStatsForwProhibits, o2->ipIfStatsForwProhibits);
11326         UPDATE_MIB(o1, udpInCksumErrs, o2->udpInCksumErrs);
11327         UPDATE_MIB(o1, udpInOverflows, o2->udpInOverflows);
11328         UPDATE_MIB(o1, rawipInOverflows, o2->rawipInOverflows);
11329         UPDATE_MIB(o1, ipIfStatsInWrongIPVersion,
11330             o2->ipIfStatsInWrongIPVersion);
11331         UPDATE_MIB(o1, ipIfStatsOutWrongIPVersion,
11332             o2->ipIfStatsInWrongIPVersion);
11333         UPDATE_MIB(o1, ipIfStatsOutSwitchIPVersion,
11334             o2->ipIfStatsOutSwitchIPVersion);
11335         UPDATE_MIB(o1, ipIfStatsHCInReceives, o2->ipIfStatsHCInReceives);
11336         UPDATE_MIB(o1, ipIfStatsHCInOctets, o2->ipIfStatsHCInOctets);
11337         UPDATE_MIB(o1, ipIfStatsHCInForwDatagrams,
11338             o2->ipIfStatsHCInForwDatagrams);
11339         UPDATE_MIB(o1, ipIfStatsHCInDelivers, o2->ipIfStatsHCInDelivers);
11340         UPDATE_MIB(o1, ipIfStatsHCOutRequests, o2->ipIfStatsHCOutRequests);
11341         UPDATE_MIB(o1, ipIfStatsHCOutForwDatagrams,
11342             o2->ipIfStatsHCOutForwDatagrams);
11343         UPDATE_MIB(o1, ipIfStatsOutFragReqds, o2->ipIfStatsOutFragReqds);
11344         UPDATE_MIB(o1, ipIfStatsHCOutTransmits, o2->ipIfStatsHCOutTransmits);
11345         UPDATE_MIB(o1, ipIfStatsHCOutOctets, o2->ipIfStatsHCOutOctets);
11346         UPDATE_MIB(o1, ipIfStatsHCInMcastPkts, o2->ipIfStatsHCInMcastPkts);
11347         UPDATE_MIB(o1, ipIfStatsHCInMcastOctets, o2->ipIfStatsHCInMcastOctets);
```

```
11348         UPDATE_MIB(o1, ipIfStatsHCOutMcastPkts, o2->ipIfStatsHCOutMcastPkts);
11349         UPDATE_MIB(o1, ipIfStatsHCOutMcastOctets,
11350             o2->ipIfStatsHCOutMcastOctets);
11351         UPDATE_MIB(o1, ipIfStatsHCInBcastPkts, o2->ipIfStatsHCInBcastPkts);
11352         UPDATE_MIB(o1, ipIfStatsHCOutBcastPkts, o2->ipIfStatsHCOutBcastPkts);
11353         UPDATE_MIB(o1, ipsecInSucceeded, o2->ipsecInSucceeded);
11354         UPDATE_MIB(o1, ipsecInFailed, o2->ipsecInFailed);
11355         UPDATE_MIB(o1, ipInCksumErrs, o2->ipInCksumErrs);
11356         UPDATE_MIB(o1, tcpInErrs, o2->tcpInErrs);
11357         UPDATE_MIB(o1, udpNoPorts, o2->udpNoPorts);
11358 }

11360 void
11361 ip_mib2_add_icmp6_stats(mib2_ipv6IfIcmpEntry_t *o1, mib2_ipv6IfIcmpEntry_t *o2)
11362 {
11363         UPDATE_MIB(o1, ipv6IfIcmpInMsgs, o2->ipv6IfIcmpInMsgs);
11364         UPDATE_MIB(o1, ipv6IfIcmpInErrors, o2->ipv6IfIcmpInErrors);
11365         UPDATE_MIB(o1, ipv6IfIcmpInDestUnreachs, o2->ipv6IfIcmpInDestUnreachs);
11366         UPDATE_MIB(o1, ipv6IfIcmpInAdminProhibs, o2->ipv6IfIcmpInAdminProhibs);
11367         UPDATE_MIB(o1, ipv6IfIcmpInTimeExcds, o2->ipv6IfIcmpInTimeExcds);
11368         UPDATE_MIB(o1, ipv6IfIcmpInParmProblems, o2->ipv6IfIcmpInParmProblems);
11369         UPDATE_MIB(o1, ipv6IfIcmpInPktTooBigs, o2->ipv6IfIcmpInPktTooBigs);
11370         UPDATE_MIB(o1, ipv6IfIcmpInEchos, o2->ipv6IfIcmpInEchos);
11371         UPDATE_MIB(o1, ipv6IfIcmpInEchoReplies, o2->ipv6IfIcmpInEchoReplies);
11372         UPDATE_MIB(o1, ipv6IfIcmpInRouterSolicits,
11373             o2->ipv6IfIcmpInRouterSolicits);
11374         UPDATE_MIB(o1, ipv6IfIcmpInRouterAdvertisements,
11375             o2->ipv6IfIcmpInRouterAdvertisements);
11376         UPDATE_MIB(o1, ipv6IfIcmpInNeighborSolicits,
11377             o2->ipv6IfIcmpInNeighborSolicits);
11378         UPDATE_MIB(o1, ipv6IfIcmpInNeighborAdvertisements,
11379             o2->ipv6IfIcmpInNeighborAdvertisements);
11380         UPDATE_MIB(o1, ipv6IfIcmpInRedirects, o2->ipv6IfIcmpInRedirects);
11381         UPDATE_MIB(o1, ipv6IfIcmpInGroupMembQueries,
11382             o2->ipv6IfIcmpInGroupMembQueries);
11383         UPDATE_MIB(o1, ipv6IfIcmpInGroupMembResponses,
11384             o2->ipv6IfIcmpInGroupMembResponses);
11385         UPDATE_MIB(o1, ipv6IfIcmpInGroupMembReductions,
11386             o2->ipv6IfIcmpInGroupMembReductions);
11387         UPDATE_MIB(o1, ipv6IfIcmpOutMsgs, o2->ipv6IfIcmpOutMsgs);
11388         UPDATE_MIB(o1, ipv6IfIcmpOutErrors, o2->ipv6IfIcmpOutErrors);
11389         UPDATE_MIB(o1, ipv6IfIcmpOutDestUnreachs,
11390             o2->ipv6IfIcmpOutDestUnreachs);
11391         UPDATE_MIB(o1, ipv6IfIcmpOutAdminProhibs,
11392             o2->ipv6IfIcmpOutAdminProhibs);
11393         UPDATE_MIB(o1, ipv6IfIcmpOutTimeExcds, o2->ipv6IfIcmpOutTimeExcds);
11394         UPDATE_MIB(o1, ipv6IfIcmpOutParmProblems,
11395             o2->ipv6IfIcmpOutParmProblems);
11396         UPDATE_MIB(o1, ipv6IfIcmpOutPktTooBigs, o2->ipv6IfIcmpOutPktTooBigs);
11397         UPDATE_MIB(o1, ipv6IfIcmpOutEchos, o2->ipv6IfIcmpOutEchos);
11398         UPDATE_MIB(o1, ipv6IfIcmpOutEchoReplies, o2->ipv6IfIcmpOutEchoReplies);
11399         UPDATE_MIB(o1, ipv6IfIcmpOutRouterSolicits,
11400             o2->ipv6IfIcmpOutRouterSolicits);
11401         UPDATE_MIB(o1, ipv6IfIcmpOutRouterAdvertisements,
11402             o2->ipv6IfIcmpOutRouterAdvertisements);
11403         UPDATE_MIB(o1, ipv6IfIcmpOutNeighborSolicits,
11404             o2->ipv6IfIcmpOutNeighborSolicits);
11405         UPDATE_MIB(o1, ipv6IfIcmpOutNeighborAdvertisements,
11406             o2->ipv6IfIcmpOutNeighborAdvertisements);
11407         UPDATE_MIB(o1, ipv6IfIcmpOutRedirects, o2->ipv6IfIcmpOutRedirects);
11408         UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembQueries,
11409             o2->ipv6IfIcmpOutGroupMembQueries);
11410         UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembResponses,
11411             o2->ipv6IfIcmpOutGroupMembResponses);
11412         UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembReductions,
11413             o2->ipv6IfIcmpOutGroupMembReductions);
```



```
11414         UPDATE_MIB(o1, ipv6IfIcmpInOverflows, o2->ipv6IfIcmpInOverflows);
11415         UPDATE_MIB(o1, ipv6IfIcmpBadHoplimit, o2->ipv6IfIcmpBadHoplimit);
11416         UPDATE_MIB(o1, ipv6IfIcmpInBadNeighborAdvertisements,
11417             o2->ipv6IfIcmpInBadNeighborAdvertisements);
11418         UPDATE_MIB(o1, ipv6IfIcmpInBadNeighborSolicitations,
11419             o2->ipv6IfIcmpInBadNeighborSolicitations);
11420         UPDATE_MIB(o1, ipv6IfIcmpInBadRedirects, o2->ipv6IfIcmpInBadRedirects);
11421         UPDATE_MIB(o1, ipv6IfIcmpInGroupMembTotal,
11422             o2->ipv6IfIcmpInGroupMembTotal);
11423         UPDATE_MIB(o1, ipv6IfIcmpInGroupMembBadQueries,
11424             o2->ipv6IfIcmpInGroupMembBadQueries);
11425         UPDATE_MIB(o1, ipv6IfIcmpInGroupMembBadReports,
11426             o2->ipv6IfIcmpInGroupMembBadReports);
11427         UPDATE_MIB(o1, ipv6IfIcmpInGroupMembOurReports,
11428             o2->ipv6IfIcmpInGroupMembOurReports);
11429 }

11431 /*
11432  * Called before the options are updated to check if this packet will
11433  * be source routed from here.
11434  * This routine assumes that the options are well formed i.e. that they
11435  * have already been checked.
11436  */
11437 boolean_t
11438 ip_source_routed(ipha_t *ipha, ip_stack_t *ipst)
11439 {
11440         ipoptp_t        opts;
11441         uchar_t         *opt;
11442         uint8_t         optval;
11443         uint8_t         optlen;
11444         ipaddr_t        dst;

11446         if (IS_SIMPLE_IPH(ipha)) {
11447                 ip2dbg(("not source routed\n"));
11448                 return (B_FALSE);
11449         }
11450         dst = ipha->ipha_dst;
11451         for (optval = ipoptp_first(&opts, ipha);
11452             optval != IPOPT_EOL;
11453             optval = ipoptp_next(&opts)) {
11454                 ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
11455                 opt = opts.ipoptp_cur;
11456                 optlen = opts.ipoptp_len;
11457                 ip2dbg(("ip_source_routed: opt %d, len %d\n",
11458                     optval, optlen));
11459                 switch (optval) {
11460                         uint32_t off;
11461                 case IPOPT_SSRR:
11462                 case IPOPT_LSRR:
11463                         /*
11464                          * If dst is one of our addresses and there are some
11465                          * entries left in the source route return (true).
11466                          */
11467                         if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
11468                                 ip2dbg(("ip_source_routed: not next"
11469                                     " source route 0x%x\n",
11470                                     ntohl(dst)));
11471                                 return (B_FALSE);
11472                         }
11473                         off = opt[IPOPT_OFFSET];
11474                         off--;
11475                         if (optlen < IP_ADDR_LEN ||
11476                             off > optlen - IP_ADDR_LEN) {
11477                                 /* End of source route */
11478                                 ip1dbg(("ip_source_routed: end of SR\n"));
11479                                 return (B_FALSE);
```

```
11480                         }
11481                         return (B_TRUE);
11482                 }
11483         }
11484         ip2dbg(("not source routed\n"));
11485         return (B_FALSE);
11486 }

11488 /*
11489  * ip_unbind is called by the transports to remove a conn from
11490  * the fanout table.
11491  */
11492 void
11493 ip_unbind(conn_t *connp)
11494 {

11496         ASSERT(!MUTEX_HELD(&connp->conn_lock));

11498         if (is_system_labeled() && connp->conn_anon_port) {
11499                 (void) tsol_mlp_anon(crgetzone(connp->conn_cred),
11500                     connp->conn_mlp_type, connp->conn_proto,
11501                     ntohs(connp->conn_lport), B_FALSE);
11502                 connp->conn_anon_port = 0;
11503         }
11504         connp->conn_mlp_type = mlptSingle;

11506         ipcl_hash_remove(connp);
11507 }

11509 /*
11510  * Used for deciding the MSS size for the upper layer. Thus
11511  * we need to check the outbound policy values in the conn.
11512  */
11513 int
11514 conn_ipsec_length(conn_t *connp)
11515 {
11516         ipsec_latch_t *ipl;

11518         ipl = connp->conn_latch;
11519         if (ipl == NULL)
11520                 return (0);

11522         if (connp->conn_ixa->ixa_ipsec_policy == NULL)
11523                 return (0);

11525         return (connp->conn_ixa->ixa_ipsec_policy->ipsp_act->ipa_ovhd);
11526 }

11528 /*
11529  * Returns an estimate of the IPsec headers size. This is used if
11530  * we don't want to call into IPsec to get the exact size.
11531  */
11532 int
11533 ipsec_out_extra_length(ip_xmit_attr_t *ixa)
11534 {
11535         ipsec_action_t *a;

11537         if (!(ixa->ixa_flags & IXAF_IPSEC_SECURE))
11538                 return (0);

11540         a = ixa->ixa_ipsec_action;
11541         if (a == NULL) {
11542                 ASSERT(ixa->ixa_ipsec_policy != NULL);
11543                 a = ixa->ixa_ipsec_policy->ipsp_act;
11544         }
11545         ASSERT(a != NULL);
```



```
11547         return (a->ipa_ovhd);
11548 }

11550 /*
11551  * If there are any source route options, return the true final
11552  * destination. Otherwise, return the destination.
11553  */
11554 ipaddr_t
11555 ip_get_dst(ipha_t *ipha)
11556 {
11557         ipoptp_t        opts;
11558         uchar_t         *opt;
11559         uint8_t         optval;
11560         uint8_t         optlen;
11561         ipaddr_t        dst;
11562         uint32_t off;

11564         dst = ipha->ipha_dst;

11566         if (IS_SIMPLE_IPH(ipha))
11567                 return (dst);

11569         for (optval = ipoptp_first(&opts, ipha);
11570             optval != IPOPT_EOL;
11571             optval = ipoptp_next(&opts)) {
11572                 opt = opts.ipoptp_cur;
11573                 optlen = opts.ipoptp_len;
11574                 ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
11575                 switch (optval) {
11576                 case IPOPT_SSRR:
11577                 case IPOPT_LSRR:
11578                         off = opt[IPOPT_OFFSET];
11579                         /*
11580                          * If one of the conditions is true, it means
11581                          * end of options and dst already has the right
11582                          * value.
11583                          */
11584                         if (!(optlen < IP_ADDR_LEN || off > optlen - 3)) {
11585                                 off = optlen - IP_ADDR_LEN;
11586                                 bcopy(&opt[off], &dst, IP_ADDR_LEN);
11587                         }
11588                         return (dst);
11589                 default:
11590                         break;
11591                 }
11592         }

11594         return (dst);
11595 }

11597 /*
11598  * Outbound IP fragmentation routine.
11599  * Assumes the caller has checked whether or not fragmentation should
11600  * be allowed. Here we copy the DF bit from the header to all the generated
11601  * fragments.
11602  */
11603 int
11604 ip_fragment_v4(mblk_t *mp_orig, nce_t *nce, iaflags_t ixaflags,
11605     uint_t pkt_len, uint32_t max_frag, uint32_t xmit_hint, zoneid_t szone,
11606     zoneid_t nolzid, pfirepostfrag_t postfragfn, uintptr_t *ixa_cookie)
11607 {
11608         int             i1;
11609         int             hdr_len;
11610         mblk_t          *hdr_mp;
11611         ipha_t          *ipha;
```

```
11612         int             ip_data_end;
11613         int             len;
11614         mblk_t          *mp = mp_orig;
11615         int             offset;
11616         ill_t           *ill = nce->nce_ill;
11617         ip_stack_t      *ipst = ill->ill_ipst;
11618         mblk_t          *carve_mp;
11619         uint32_t        frag_flag;
11620         uint_t          priority = mp->b_band;
11621         int             error = 0;

11623         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragReqds);

11625         if (pkt_len != msgdsize(mp)) {
11626                 ip0dbg(("Packet length mismatch: %d, %ld\n",
11627                     pkt_len, msgdsize(mp)));
11628                 freemsg(mp);
11629                 return (EINVAL);
11630         }

11632         if (max_frag == 0) {
11633                 ip1dbg(("ip_fragment_v4: max_frag is zero. Dropping packet\n"));
11634                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11635                 ip_drop_output("FragFails: zero max_frag", mp, ill);
11636                 freemsg(mp);
11637                 return (EINVAL);
11638         }

11640         ASSERT(MBLKL(mp) >= sizeof (ipha_t));
11641         ipha = (ipha_t *)mp->b_rptr;
11642         ASSERT(ntohs(ipha->ipha_length) == pkt_len);
11643         frag_flag = ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_DF;

11645         /*
11646          * Establish the starting offset.  May not be zero if we are fragging
11647          * a fragment that is being forwarded.
11648          */
11649         offset = ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_OFFSET;

11651         /* TODO why is this test needed? */
11652         if (((max_frag - ntohs(ipha->ipha_length)) & ~7) < 8) {
11653                 /* TODO: notify ulp somehow */
11654                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11655                 ip_drop_output("FragFails: bad starting offset", mp, ill);
11656                 freemsg(mp);
11657                 return (EINVAL);
11658         }

11660         hdr_len = IPH_HDR_LENGTH(ipha);
11661         ipha->ipha_hdr_checksum = 0;

11663         /*
11664          * Establish the number of bytes maximum per frag, after putting
11665          * in the header.
11666          */
11667         len = (max_frag - hdr_len) & ~7;

11669         /* Get a copy of the header for the trailing frags */
11670         hdr_mp = ip_fragment_copyhdr((uchar_t *)ipha, hdr_len, offset, ipst,
11671             mp);
11672         if (hdr_mp == NULL) {
11673                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11674                 ip_drop_output("FragFails: no hdr_mp", mp, ill);
11675                 freemsg(mp);
11676                 return (ENOBUFS);
11677         }
```



```
11679         /* Store the starting offset, with the MoreFrags flag. */
11680         i1 = offset | IPH_MF | frag_flag;
11681         ipha->ipha_fragment_offset_and_flags = htons((uint16_t)i1);

11683         /* Establish the ending byte offset, based on the starting offset. */
11684         offset <<= 3;
11685         ip_data_end = offset + ntohs(ipha->ipha_length) - hdr_len;

11687         /* Store the length of the first fragment in the IP header. */
11688         i1 = len + hdr_len;
11689         ASSERT(i1 <= IP_MAXPACKET);
11690         ipha->ipha_length = htons((uint16_t)i1);

11692         /*
11693          * Compute the IP header checksum for the first frag.  We have to
11694          * watch out that we stop at the end of the header.
11695          */
11696         ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

11698         /*
11699          * Now carve off the first frag.  Note that this will include the
11700          * original IP header.
11701          */
11702         if (!(mp = ip_carve_mp(&mp_orig, i1))) {
11703                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11704                 ip_drop_output("FragFails: could not carve mp", mp_orig, ill);
11705                 freeb(hdr_mp);
11706                 freemsg(mp_orig);
11707                 return (ENOBUFS);
11708         }

11710         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragCreates);

11712         error = postfragfn(mp, nce, ixaflags, i1, xmit_hint, szone, nolzid,
11713             ixa_cookie);
11714         if (error != 0 && error != EWOULDBLOCK) {
11715                 /* No point in sending the other fragments */
11716                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11717                 ip_drop_output("FragFails: postfragfn failed", mp_orig, ill);
11718                 freeb(hdr_mp);
11719                 freemsg(mp_orig);
11720                 return (error);
11721         }

11723         /* No need to redo state machine in loop */
11724         ixaflags &= ~IXAF_REACH_CONF;

11726         /* Advance the offset to the second frag starting point. */
11727         offset += len;
11728         /*
11729          * Update hdr_len from the copied header - there might be less options
11730          * in the later fragments.
11731          */
11732         hdr_len = IPH_HDR_LENGTH(hdr_mp->b_rptr);
11733         /* Loop until done. */
11734         for (;;) {
11735                 uint16_t        offset_and_flags;
11736                 uint16_t        ip_len;

11738                 if (ip_data_end - offset > len) {
11739                         /*
11740                          * Carve off the appropriate amount from the original
11741                          * datagram.
11742                          */
11743                         if (!(carve_mp = ip_carve_mp(&mp_orig, len))) {
```

```
11744                                   mp = NULL;
11745                                   break;
11746                           }
11747                           /*
11748                            * More frags after this one.  Get another copy
11749                            * of the header.
11750                            */
11751                           if (carve_mp->b_datap->db_ref == 1 &&
11752                               hdr_mp->b_wptr - hdr_mp->b_rptr <
11753                               carve_mp->b_rptr - carve_mp->b_datap->db_base) {
11754                                   /* Inline IP header */
11755                                   carve_mp->b_rptr -= hdr_mp->b_wptr -
11756                                       hdr_mp->b_rptr;
11757                                   bcopy(hdr_mp->b_rptr, carve_mp->b_rptr,
11758                                       hdr_mp->b_wptr - hdr_mp->b_rptr);
11759                                   mp = carve_mp;
11760                           } else {
11761                                   if (!(mp = copyb(hdr_mp))) {
11762                                           freemsg(carve_mp);
11763                                           break;
11764                                   }
11765                                   /* Get priority marking, if any. */
11766                                   mp->b_band = priority;
11767                                   mp->b_cont = carve_mp;
11768                           }
11769                           ipha = (ipha_t *)mp->b_rptr;
11770                           offset_and_flags = IPH_MF;
11771                   } else {
11772                           /*
11773                            * Last frag.  Consume the header. Set len to
11774                            * the length of this last piece.
11775                            */
11776                           len = ip_data_end - offset;

11778                           /*
11779                            * Carve off the appropriate amount from the original
11780                            * datagram.
11781                            */
11782                           if (!(carve_mp = ip_carve_mp(&mp_orig, len))) {
11783                                   mp = NULL;
11784                                   break;
11785                           }
11786                           if (carve_mp->b_datap->db_ref == 1 &&
11787                               hdr_mp->b_wptr - hdr_mp->b_rptr <
11788                               carve_mp->b_rptr - carve_mp->b_datap->db_base) {
11789                                   /* Inline IP header */
11790                                   carve_mp->b_rptr -= hdr_mp->b_wptr -
11791                                       hdr_mp->b_rptr;
11792                                   bcopy(hdr_mp->b_rptr, carve_mp->b_rptr,
11793                                       hdr_mp->b_wptr - hdr_mp->b_rptr);
11794                                   mp = carve_mp;
11795                                   freeb(hdr_mp);
11796                                   hdr_mp = mp;
11797                           } else {
11798                                   mp = hdr_mp;
11799                                   /* Get priority marking, if any. */
11800                                   mp->b_band = priority;
11801                                   mp->b_cont = carve_mp;
11802                           }
11803                           ipha = (ipha_t *)mp->b_rptr;
11804                           /* A frag of a frag might have IPH_MF non-zero */
11805                           offset_and_flags =
11806                               ntohs(ipha->ipha_fragment_offset_and_flags) &
11807                               IPH_MF;
11808                   }
11809                   offset_and_flags |= (uint16_t)(offset >> 3);
```



```
11810                   offset_and_flags |= (uint16_t)frag_flag;
11811                   /* Store the offset and flags in the IP header. */
11812                   ipha->ipha_fragment_offset_and_flags = htons(offset_and_flags);

11814                   /* Store the length in the IP header. */
11815                   ip_len = (uint16_t)(len + hdr_len);
11816                   ipha->ipha_length = htons(ip_len);

11818                   /*
11819                    * Set the IP header checksum.  Note that mp is just
11820                    * the header, so this is easy to pass to ip_csum.
11821                    */
11822                   ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

11824                   BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragCreates);

11826                   error = postfragfn(mp, nce, ixaflags, ip_len, xmit_hint, szone,
11827                       nolzid, ixa_cookie);
11828                   /* All done if we just consumed the hdr_mp. */
11829                   if (mp == hdr_mp) {
11830                           BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragOKs);
11831                           return (error);
11832                   }
11833                   if (error != 0 && error != EWOULDBLOCK) {
11834                           DTRACE_PROBE2(ip__xmit__frag__fail, ill_t *, ill,
11835                               mblk_t *, hdr_mp);
11836                           /* No point in sending the other fragments */
11837                           break;
11838                   }

11840                   /* Otherwise, advance and loop. */
11841                   offset += len;
11842           }
11843           /* Clean up following allocation failure. */
11844           BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11845           ip_drop_output("FragFails: loop ended", NULL, ill);
11846           if (mp != hdr_mp)
11847                   freeb(hdr_mp);
11848           if (mp != mp_orig)
11849                   freemsg(mp_orig);
11850           return (error);
11851   }

11853   /*
11854    * Copy the header plus those options which have the copy bit set
11855    */
11856   static mblk_t *
11857   ip_fragment_copyhdr(uchar_t *rptr, int hdr_len, int offset, ip_stack_t *ipst,
11858       mblk_t *src)
11859   {
11860           mblk_t  *mp;
11861           uchar_t *up;

11863           /*
11864            * Quick check if we need to look for options without the copy bit
11865            * set
11866            */
11867           mp = allocb_tmpl(ipst->ips_ip_wroff_extra + hdr_len, src);
11868           if (!mp)
11869                   return (mp);
11870           mp->b_rptr += ipst->ips_ip_wroff_extra;
11871           if (hdr_len == IP_SIMPLE_HDR_LENGTH || offset != 0) {
11872                   bcopy(rptr, mp->b_rptr, hdr_len);
11873                   mp->b_wptr += hdr_len + ipst->ips_ip_wroff_extra;
11874                   return (mp);
11875           }
```

```
11876         up  = mp->b_rptr;
11877         bcopy(rptr, up, IP_SIMPLE_HDR_LENGTH);
11878         up += IP_SIMPLE_HDR_LENGTH;
11879         rptr += IP_SIMPLE_HDR_LENGTH;
11880         hdr_len -= IP_SIMPLE_HDR_LENGTH;
11881         while (hdr_len > 0) {
11882                 uint32_t optval;
11883                 uint32_t optlen;

11885                 optval = *rptr;
11886                 if (optval == IPOPT_EOL)
11887                         break;
11888                 if (optval == IPOPT_NOP)
11889                         optlen = 1;
11890                 else
11891                         optlen = rptr[1];
11892                 if (optval & IPOPT_COPY) {
11893                         bcopy(rptr, up, optlen);
11894                         up += optlen;
11895                 }
11896                 rptr += optlen;
11897                 hdr_len -= optlen;
11898         }
11899         /*
11900          * Make sure that we drop an even number of words by filling
11901          * with EOL to the next word boundary.
11902          */
11903         for (hdr_len = up - (mp->b_rptr + IP_SIMPLE_HDR_LENGTH);
11904             hdr_len & 0x3; hdr_len++)
11905                 *up++ = IPOPT_EOL;
11906         mp->b_wptr = up;
11907         /* Update header length */
11908         mp->b_rptr[0] = (uint8_t)((IP_VERSION << 4) | ((up - mp->b_rptr) >> 2));
11909         return (mp);
11910 }

11912 /*
11913  * Update any source route, record route, or timestamp options when
11914  * sending a packet back to ourselves.
11915  * Check that we are at end of strict source route.
11916  * The options have been sanity checked by ip_output_options().
11917  */
11918 void
11919 ip_output_local_options(ipha_t *ipha, ip_stack_t *ipst)
11920 {
11921         ipoptp_t        opts;
11922         uchar_t         *opt;
11923         uint8_t         optval;
11924         uint8_t         optlen;
11925         ipaddr_t        dst;
11926         uint32_t        ts;
11927         timestruc_t     now;

11929         for (optval = ipoptp_first(&opts, ipha);
11930             optval != IPOPT_EOL;
11931             optval = ipoptp_next(&opts)) {
11932                 opt = opts.ipoptp_cur;
11933                 optlen = opts.ipoptp_len;
11934                 ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
11935                 switch (optval) {
11936                         uint32_t off;
11937                 case IPOPT_SSRR:
11938                 case IPOPT_LSRR:
11939                         off = opt[IPOPT_OFFSET];
11940                         off--;
11941                         if (optlen < IP_ADDR_LEN ||
```



```
11942                             off > optlen - IP_ADDR_LEN) {
11943                                 /* End of source route */
11944                                 break;
11945                         }
11946                         /*
11947                          * This will only happen if two consecutive entries
11948                          * in the source route contains our address or if
11949                          * it is a packet with a loose source route which
11950                          * reaches us before consuming the whole source route
11951                          */

11953                         if (optval == IPOPT_SSRR) {
11954                                 return;
11955                         }
11956                         /*
11957                          * Hack: instead of dropping the packet truncate the
11958                          * source route to what has been used by filling the
11959                          * rest with IPOPT_NOP.
11960                          */
11961                         opt[IPOPT_OLEN] = (uint8_t)off;
11962                         while (off < optlen) {
11963                                 opt[off++] = IPOPT_NOP;
11964                         }
11965                         break;
11966                 case IPOPT_RR:
11967                         off = opt[IPOPT_OFFSET];
11968                         off--;
11969                         if (optlen < IP_ADDR_LEN ||
11970                             off > optlen - IP_ADDR_LEN) {
11971                                 /* No more room - ignore */
11972                                 ip1dbg((
11973                                     "ip_output_local_options: end of RR\n"));
11974                                 break;
11975                         }
11976                         dst = htonl(INADDR_LOOPBACK);
11977                         bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
11978                         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
11979                         break;
11980                 case IPOPT_TS:
11981                         /* Insert timestamp if there is romm */
11982                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
11983                         case IPOPT_TS_TSONLY:
11984                                 off = IPOPT_TS_TIMELEN;
11985                                 break;
11986                         case IPOPT_TS_PRESPEC:
11987                         case IPOPT_TS_PRESPEC_RFC791:
11988                                 /* Verify that the address matched */
11989                                 off = opt[IPOPT_OFFSET] - 1;
11990                                 bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
11991                                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
11992                                         /* Not for us */
11993                                         break;
11994                                 }
11995                                 /* FALLTHRU */
11996                         case IPOPT_TS_TSANDADDR:
11997                                 off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
11998                                 break;
11999                         default:
12000                                 /*
12001                                  * ip_*put_options should have already
12002                                  * dropped this packet.
12003                                  */
12004                                 cmn_err(CE_PANIC, "ip_output_local_options: "
12005                                     "unknown IT - bug in ip_output_options?\n");
12006                                 return; /* Keep "lint" happy */
12007                         }
```

```
12008                         if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
12009                                 /* Increase overflow counter */
12010                                 off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
12011                                 opt[IPOPT_POS_OV_FLG] = (uint8_t)
12012                                     (opt[IPOPT_POS_OV_FLG] & 0x0F) |
12013                                     (off << 4);
12014                                 break;
12015                         }
12016                         off = opt[IPOPT_OFFSET] - 1;
12017                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
12018                         case IPOPT_TS_PRESPEC:
12019                         case IPOPT_TS_PRESPEC_RFC791:
12020                         case IPOPT_TS_TSANDADDR:
12021                                 dst = htonl(INADDR_LOOPBACK);
12022                                 bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
12023                                 opt[IPOPT_OFFSET] += IP_ADDR_LEN;
12024                                 /* FALLTHRU */
12025                         case IPOPT_TS_TSONLY:
12026                                 off = opt[IPOPT_OFFSET] - 1;
12027                                 /* Compute # of milliseconds since midnight */
12028                                 gethrestime(&now);
12029                                 ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
12030                                     now.tv_nsec / (NANOSEC / MILLISEC);
12031                                 bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
12032                                 opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
12033                                 break;
12034                         }
12035                         break;
12036                 }
12037         }
12038 }

12040 /*
12041  * Prepend an M_DATA fastpath header, and if none present prepend a
12042  * DL_UNITDATA_REQ. Frees the mblk on failure.
12043  *
12044  * nce_dlur_mp and nce_fp_mp can not disappear once they have been set.
12045  * If there is a change to them, the nce will be deleted (condemned) and
12046  * a new nce_t will be created when packets are sent. Thus we need no locks
12047  * to access those fields.
12048  *
12049  * We preserve b_band to support IPQoS. If a DL_UNITDATA_REQ is prepended
12050  * we place b_band in dl_priority.dl_max.
12051  */
12052 static mblk_t *
12053 ip_xmit_attach_llhdr(mblk_t *mp, nce_t *nce)
12054 {
12055         uint_t  hlen;
12056         mblk_t *mp1;
12057         uint_t  priority;
12058         uchar_t *rptr;

12060         rptr = mp->b_rptr;

12062         ASSERT(DB_TYPE(mp) == M_DATA);
12063         priority = mp->b_band;

12065         ASSERT(nce != NULL);
12066         if ((mp1 = nce->nce_fp_mp) != NULL) {
12067                 hlen = MBLKL(mp1);
12068                 /*
12069                  * Check if we have enough room to prepend fastpath
12070                  * header
12071                  */
12072                 if (hlen != 0 && (rptr - mp->b_datap->db_base) >= hlen) {
12073                         rptr -= hlen;
```



```
12074                         bcopy(mp1->b_rptr, rptr, hlen);
12075                         /*
12076                          * Set the b_rptr to the start of the link layer
12077                          * header
12078                          */
12079                         mp->b_rptr = rptr;
12080                         return (mp);
12081                 }
12082                 mp1 = copyb(mp1);
12083                 if (mp1 == NULL) {
12084                         ill_t *ill = nce->nce_ill;

12086                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12087                         ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12088                         freemsg(mp);
12089                         return (NULL);
12090                 }
12091                 mp1->b_band = priority;
12092                 mp1->b_cont = mp;
12093                 DB_CKSUMSTART(mp1) = DB_CKSUMSTART(mp);
12094                 DB_CKSUMSTUFF(mp1) = DB_CKSUMSTUFF(mp);
12095                 DB_CKSUMEND(mp1) = DB_CKSUMEND(mp);
12096                 DB_CKSUMFLAGS(mp1) = DB_CKSUMFLAGS(mp);
12097                 DB_LSOMSS(mp1) = DB_LSOMSS(mp);
12098                 DTRACE_PROBE1(ip__xmit__copyb, (mblk_t *), mp1);
12099                 /*
12100                  * XXX disable ICK_VALID and compute checksum
12101                  * here; can happen if nce_fp_mp changes and
12102                  * it can't be copied now due to insufficient
12103                  * space. (unlikely, fp mp can change, but it
12104                  * does not increase in length)
12105                  */
12106                 return (mp1);
12107         }
12108         mp1 = copyb(nce->nce_dlur_mp);

12110         if (mp1 == NULL) {
12111                 ill_t *ill = nce->nce_ill;

12113                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12114                 ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12115                 freemsg(mp);
12116                 return (NULL);
12117         }
12118         mp1->b_cont = mp;
12119         if (priority != 0) {
12120                 mp1->b_band = priority;
12121                 ((dl_unitdata_req_t *)(mp1->b_rptr))->dl_priority.dl_max =
12122                     priority;
12123         }
12124         return (mp1);
12125 #undef rptr
12126 }

12128 /*
12129  * Finish the outbound IPsec processing. This function is called from
12130  * ipsec_out_process() if the IPsec packet was processed
12131  * synchronously, or from {ah,esp}_kcf_callback_outbound() if it was processed
12132  * asynchronously.
12133  *
12134  * This is common to IPv4 and IPv6.
12135  */
12136 int
12137 ip_output_post_ipsec(mblk_t *mp, ip_xmit_attr_t *ixa)
12138 {
12139         iaflags_t       ixaflags = ixa->ixa_flags;
```

```
12140         uint_t          pktlen;


12143         /* AH/ESP don't update ixa_pktlen when they modify the packet */
12144         if (ixaflags & IXAF_IS_IPV4) {
12145                 ipha_t          *ipha = (ipha_t *)mp->b_rptr;

12147                 ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);
12148                 pktlen = ntohs(ipha->ipha_length);
12149         } else {
12150                 ip6_t           *ip6h = (ip6_t *)mp->b_rptr;

12152                 ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);
12153                 pktlen = ntohs(ip6h->ip6_plen) + IPV6_HDR_LEN;
12154         }

12156         /*
12157          * We release any hard reference on the SAs here to make
12158          * sure the SAs can be garbage collected. ipsr_sa has a soft reference
12159          * on the SAs.
12160          * If in the future we want the hard latching of the SAs in the
12161          * ip_xmit_attr_t then we should remove this.
12162          */
12163         if (ixa->ixa_ipsec_esp_sa != NULL) {
12164                 IPSA_REFRELE(ixa->ixa_ipsec_esp_sa);
12165                 ixa->ixa_ipsec_esp_sa = NULL;
12166         }
12167         if (ixa->ixa_ipsec_ah_sa != NULL) {
12168                 IPSA_REFRELE(ixa->ixa_ipsec_ah_sa);
12169                 ixa->ixa_ipsec_ah_sa = NULL;
12170         }

12172         /* Do we need to fragment? */
12173         if ((ixa->ixa_flags & IXAF_IPV6_ADD_FRAGHDR) ||
12174             pktlen > ixa->ixa_fragsize) {
12175                 if (ixaflags & IXAF_IS_IPV4) {
12176                         ASSERT(!(ixa->ixa_flags & IXAF_IPV6_ADD_FRAGHDR));
12177                         /*
12178                          * We check for the DF case in ipsec_out_process
12179                          * hence this only handles the non-DF case.
12180                          */
12181                         return (ip_fragment_v4(mp, ixa->ixa_nce, ixa->ixa_flags,
12182                             pktlen, ixa->ixa_fragsize,
12183                             ixa->ixa_xmit_hint, ixa->ixa_zoneid,
12184                             ixa->ixa_no_loop_zoneid, ixa->ixa_postfragfn,
12185                             &ixa->ixa_cookie));
12186                 } else {
12187                         mp = ip_fraghdr_add_v6(mp, ixa->ixa_ident, ixa);
12188                         if (mp == NULL) {
12189                                 /* MIB and ip_drop_output already done */
12190                                 return (ENOMEM);
12191                         }
12192                         pktlen += sizeof (ip6_frag_t);
12193                         if (pktlen > ixa->ixa_fragsize) {
12194                                 return (ip_fragment_v6(mp, ixa->ixa_nce,
12195                                     ixa->ixa_flags, pktlen,
12196                                     ixa->ixa_fragsize, ixa->ixa_xmit_hint,
12197                                     ixa->ixa_zoneid, ixa->ixa_no_loop_zoneid,
12198                                     ixa->ixa_postfragfn, &ixa->ixa_cookie));
12199                         }
12200                 }
12201         }
12202         return ((ixa->ixa_postfragfn)(mp, ixa->ixa_nce, ixa->ixa_flags,
12203             pktlen, ixa->ixa_xmit_hint, ixa->ixa_zoneid,
12204             ixa->ixa_no_loop_zoneid, NULL));
12205 }
```



```
12207 /*
12208  * Finish the inbound IPsec processing. This function is called from
12209  * ipsec_out_process() if the IPsec packet was processed
12210  * synchronously, or from {ah,esp}_kcf_callback_outbound() if it was processed
12211  * asynchronously.
12212  *
12213  * This is common to IPv4 and IPv6.
12214  */
12215 void
12216 ip_input_post_ipsec(mblk_t *mp, ip_recv_attr_t *ira)
12217 {
12218         iaflags_t       iraflags = ira->ira_flags;

12220         /* Length might have changed */
12221         if (iraflags & IRAF_IS_IPV4) {
12222                 ipha_t          *ipha = (ipha_t *)mp->b_rptr;

12224                 ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);
12225                 ira->ira_pktlen = ntohs(ipha->ipha_length);
12226                 ira->ira_ip_hdr_length = IPH_HDR_LENGTH(ipha);
12227                 ira->ira_protocol = ipha->ipha_protocol;

12229                 ip_fanout_v4(mp, ipha, ira);
12230         } else {
12231                 ip6_t           *ip6h = (ip6_t *)mp->b_rptr;
12232                 uint8_t         *nexthdrp;

12234                 ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);
12235                 ira->ira_pktlen = ntohs(ip6h->ip6_plen) + IPV6_HDR_LEN;
12236                 if (!ip_hdr_length_nexthdr_v6(mp, ip6h, &ira->ira_ip_hdr_length,
12237                     &nexthdrp)) {
12238                         /* Malformed packet */
12239                         BUMP_MIB(ira->ira_ill->ill_ip_mib, ipIfStatsInDiscards);
12240                         ip_drop_input("ipIfStatsInDiscards", mp, ira->ira_ill);
12241                         freemsg(mp);
12242                         return;
12243                 }
12244                 ira->ira_protocol = *nexthdrp;
12245                 ip_fanout_v6(mp, ip6h, ira);
12246         }
12247 }

12249 /*
12250  * Select which AH & ESP SA's to use (if any) for the outbound packet.
12251  *
12252  * If this function returns B_TRUE, the requested SA's have been filled
12253  * into the ixa_ipsec_*_sa pointers.
12254  *
12255  * If the function returns B_FALSE, the packet has been "consumed", most
12256  * likely by an ACQUIRE sent up via PF_KEY to a key management daemon.
12257  *
12258  * The SA references created by the protocol-specific "select"
12259  * function will be released in ip_output_post_ipsec.
12260  */
12261 static boolean_t
12262 ipsec_out_select_sa(mblk_t *mp, ip_xmit_attr_t *ixa)
12263 {
12264         boolean_t need_ah_acquire = B_FALSE, need_esp_acquire = B_FALSE;
12265         ipsec_policy_t *pp;
12266         ipsec_action_t *ap;

12268         ASSERT(ixa->ixa_flags & IXAF_IPSEC_SECURE);
12269         ASSERT((ixa->ixa_ipsec_policy != NULL) ||
12270             (ixa->ixa_ipsec_action != NULL));
```

```
12272         ap = ixa->ixa_ipsec_action;
12273         if (ap == NULL) {
12274                 pp = ixa->ixa_ipsec_policy;
12275                 ASSERT(pp != NULL);
12276                 ap = pp->ipsp_act;
12277                 ASSERT(ap != NULL);
12278         }

12280         /*
12281          * We have an action.  now, let's select SA's.
12282          * A side effect of setting ixa_ipsec_*_sa is that it will
12283          * be cached in the conn_t.
12284          */
12285         if (ap->ipa_want_esp) {
12286                 if (ixa->ixa_ipsec_esp_sa == NULL) {
12287                         need_esp_acquire = !ipsec_outbound_sa(mp, ixa,
12288                             IPPROTO_ESP);
12289                 }
12290                 ASSERT(need_esp_acquire || ixa->ixa_ipsec_esp_sa != NULL);
12291         }

12293         if (ap->ipa_want_ah) {
12294                 if (ixa->ixa_ipsec_ah_sa == NULL) {
12295                         need_ah_acquire = !ipsec_outbound_sa(mp, ixa,
12296                             IPPROTO_AH);
12297                 }
12298                 ASSERT(need_ah_acquire || ixa->ixa_ipsec_ah_sa != NULL);
12299                 /*
12300                  * The ESP and AH processing order needs to be preserved
12301                  * when both protocols are required (ESP should be applied
12302                  * before AH for an outbound packet). Force an ESP ACQUIRE
12303                  * when both ESP and AH are required, and an AH ACQUIRE
12304                  * is needed.
12305                  */
12306                 if (ap->ipa_want_esp && need_ah_acquire)
12307                         need_esp_acquire = B_TRUE;
12308         }

12310         /*
12311          * Send an ACQUIRE (extended, regular, or both) if we need one.
12312          * Release SAs that got referenced, but will not be used until we
12313          * acquire _all_ of the SAs we need.
12314          */
12315         if (need_ah_acquire || need_esp_acquire) {
12316                 if (ixa->ixa_ipsec_ah_sa != NULL) {
12317                         IPSA_REFRELE(ixa->ixa_ipsec_ah_sa);
12318                         ixa->ixa_ipsec_ah_sa = NULL;
12319                 }
12320                 if (ixa->ixa_ipsec_esp_sa != NULL) {
12321                         IPSA_REFRELE(ixa->ixa_ipsec_esp_sa);
12322                         ixa->ixa_ipsec_esp_sa = NULL;
12323                 }

12325                 sadb_acquire(mp, ixa, need_ah_acquire, need_esp_acquire);
12326                 return (B_FALSE);
12327         }

12329         return (B_TRUE);
12330 }

12332 /*
12333  * Handle IPsec output processing.
12334  * This function is only entered once for a given packet.
12335  * We try to do things synchronously, but if we need to have user-level
12336  * set up SAs, or ESP or AH uses asynchronous kEF, then the operation
12337  * will be completed
```



```
12338  *   - when the SAs are added in esp_add_sa_finish/ah_add_sa_finish
12339  *   - when asynchronous ESP is done it will do AH
12340  *
12341  * In all cases we come back in ip_output_post_ipsec() to fragment and
12342  * send out the packet.
12343  */
12344 int
12345 ipsec_out_process(mblk_t *mp, ip_xmit_attr_t *ixa)
12346 {
12347         ill_t           *ill = ixa->ixa_nce->nce_ill;
12348         ip_stack_t      *ipst = ixa->ixa_ipst;
12349         ipsec_stack_t   *ipss;
12350         ipsec_policy_t  *pp;
12351         ipsec_action_t  *ap;

12353         ASSERT(ixa->ixa_flags & IXAF_IPSEC_SECURE);

12355         ASSERT((ixa->ixa_ipsec_policy != NULL) ||
12356             (ixa->ixa_ipsec_action != NULL));

12358         ipss = ipst->ips_netstack->netstack_ipsec;
12359         if (!ipsec_loaded(ipss)) {
12360                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12361                 ip_drop_packet(mp, B_TRUE, ill,
12362                     DROPPER(ipss, ipds_ip_ipsec_not_loaded),
12363                     &ipss->ipsec_dropper);
12364                 return (ENOTSUP);
12365         }

12367         ap = ixa->ixa_ipsec_action;
12368         if (ap == NULL) {
12369                 pp = ixa->ixa_ipsec_policy;
12370                 ASSERT(pp != NULL);
12371                 ap = pp->ipsp_act;
12372                 ASSERT(ap != NULL);
12373         }

12375         /* Handle explicit drop action and bypass. */
12376         switch (ap->ipa_act.ipa_type) {
12377         case IPSEC_ACT_DISCARD:
12378         case IPSEC_ACT_REJECT:
12379                 ip_drop_packet(mp, B_FALSE, ill,
12380                     DROPPER(ipss, ipds_spd_explicit), &ipss->ipsec_spd_dropper);
12381                 return (EHOSTUNREACH);  /* IPsec policy failure */
12382         case IPSEC_ACT_BYPASS:
12383                 return (ip_output_post_ipsec(mp, ixa));
12384         }

12386         /*
12387          * The order of processing is first insert a IP header if needed.
12388          * Then insert the ESP header and then the AH header.
12389          */
12390         if ((ixa->ixa_flags & IXAF_IS_IPV4) && ap->ipa_want_se) {
12391                 /*
12392                  * First get the outer IP header before sending
12393                  * it to ESP.
12394                  */
12395                 ipha_t *oipha, *iipha;
12396                 mblk_t *outer_mp, *inner_mp;

12398                 if ((outer_mp = allocb(sizeof (ipha_t), BPRI_HI)) == NULL) {
12399                         (void) mi_strlog(ill->ill_rq, 0,
12400                             SL_ERROR|SL_TRACE|SL_CONSOLE,
12401                             "ipsec_out_process: "
12402                             "Self-Encapsulation failed: Out of memory\n");
12403                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
```

```
12404                         ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12405                         freemsg(mp);
12406                         return (ENOBUFS);
12407                 }
12408                 inner_mp = mp;
12409                 ASSERT(inner_mp->b_datap->db_type == M_DATA);
12410                 oipha = (ipha_t *)outer_mp->b_rptr;
12411                 iipha = (ipha_t *)inner_mp->b_rptr;
12412                 *oipha = *iipha;
12413                 outer_mp->b_wptr += sizeof (ipha_t);
12414                 oipha->ipha_length = htons(ntohs(iipha->ipha_length) +
12415                     sizeof (ipha_t));
12416                 oipha->ipha_protocol = IPPROTO_ENCAP;
12417                 oipha->ipha_version_and_hdr_length =
12418                     IP_SIMPLE_HDR_VERSION;
12419                 oipha->ipha_hdr_checksum = 0;
12420                 oipha->ipha_hdr_checksum = ip_csum_hdr(oipha);
12421                 outer_mp->b_cont = inner_mp;
12422                 mp = outer_mp;

12424                 ixa->ixa_flags |= IXAF_IPSEC_TUNNEL;
12425         }

12427         /* If we need to wait for a SA then we can't return any errno */
12428         if (((ap->ipa_want_ah && (ixa->ixa_ipsec_ah_sa == NULL)) ||
12429             (ap->ipa_want_esp && (ixa->ixa_ipsec_esp_sa == NULL))) &&
12430             !ipsec_out_select_sa(mp, ixa))
12431                 return (0);

12433         /*
12434          * By now, we know what SA's to use.  Toss over to ESP & AH
12435          * to do the heavy lifting.
12436          */
12437         if (ap->ipa_want_esp) {
12438                 ASSERT(ixa->ixa_ipsec_esp_sa != NULL);

12440                 mp = ixa->ixa_ipsec_esp_sa->ipsa_output_func(mp, ixa);
12441                 if (mp == NULL) {
12442                         /*
12443                          * Either it failed or is pending. In the former case
12444                          * ipIfStatsInDiscards was increased.
12445                          */
12446                         return (0);
12447                 }
12448         }

12450         if (ap->ipa_want_ah) {
12451                 ASSERT(ixa->ixa_ipsec_ah_sa != NULL);

12453                 mp = ixa->ixa_ipsec_ah_sa->ipsa_output_func(mp, ixa);
12454                 if (mp == NULL) {
12455                         /*
12456                          * Either it failed or is pending. In the former case
12457                          * ipIfStatsInDiscards was increased.
12458                          */
12459                         return (0);
12460                 }
12461         }
12462         /*
12463          * We are done with IPsec processing. Send it over
12464          * the wire.
12465          */
12466         return (ip_output_post_ipsec(mp, ixa));
12467 }

12469 /*
```



```
12470  * ioctls that go through a down/up sequence may need to wait for the down
12471  * to complete. This involves waiting for the ire and ipif refcnts to go down
12472  * to zero. Subsequently the ioctl is restarted from ipif_ill_refrele_tail.
12473  */
12474 /* ARGSUSED */
12475 void
12476 ip_reprocess_ioctl(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
12477 {
12478         struct iocblk *iocp;
12479         mblk_t *mp1;
12480         ip_ioctl_cmd_t *ipip;
12481         int err;
12482         sin_t   *sin;
12483         struct lifreq *lifr;
12484         struct ifreq *ifr;

12486         iocp = (struct iocblk *)mp->b_rptr;
12487         ASSERT(ipsq != NULL);
12488         /* Existence of mp1 verified in ip_wput_nondata */
12489         mp1 = mp->b_cont->b_cont;
12490         ipip = ip_sioctl_lookup(iocp->ioc_cmd);
12491         if (ipip->ipi_cmd == SIOCSLIFNAME || ipip->ipi_cmd == IF_UNITSEL) {
12492                 /*
12493                  * Special case where ipx_current_ipif is not set:
12494                  * ill_phyint_reinit merged the v4 and v6 into a single ipsq.
12495                  * We are here as were not able to complete the operation in
12496                  * ipif_set_values because we could not become exclusive on
12497                  * the new ipsq.
12498                  */
12499                 ill_t *ill = q->q_ptr;
12500                 ipsq_current_start(ipsq, ill->ill_ipif, ipip->ipi_cmd);
12501         }
12502         ASSERT(ipsq->ipsq_xop->ipx_current_ipif != NULL);

12504         if (ipip->ipi_cmd_type == IF_CMD) {
12505                 /* This a old style SIOC[GS]IF* command */
12506                 ifr = (struct ifreq *)mp1->b_rptr;
12507                 sin = (sin_t *)&ifr->ifr_addr;
12508         } else if (ipip->ipi_cmd_type == LIF_CMD) {
12509                 /* This a new style SIOC[GS]LIF* command */
12510                 lifr = (struct lifreq *)mp1->b_rptr;
12511                 sin = (sin_t *)&lifr->lifr_addr;
12512         } else {
12513                 sin = NULL;
12514         }

12516         err = (*ipip->ipi_func_restart)(ipsq->ipsq_xop->ipx_current_ipif, sin,
12517             q, mp, ipip, mp1->b_rptr);

12519         DTRACE_PROBE4(ipif__ioctl, char *, "ip_reprocess_ioctl finish",
12520             int, ipip->ipi_cmd,
12521             ill_t *, ipsq->ipsq_xop->ipx_current_ipif->ipif_ill,
12522             ipif_t *, ipsq->ipsq_xop->ipx_current_ipif);

12524         ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), ipsq);
12525 }

12527 /*
12528  * ioctl processing
12529  *
12530  * ioctl processing starts with ip_sioctl_copyin_setup(), which looks up
12531  * the ioctl command in the ioctl tables, determines the copyin data size
12532  * from the ipi_copyin_size field, and does an mi_copyin() of that size.
12533  *
12534  * ioctl processing then continues when the M_IOCDATA makes its way down to
12535  * ip_wput_nondata().  The ioctl is looked up again in the ioctl table, its
```

```
12536  * associated 'conn' is refheld till the end of the ioctl and the general
12537  * ioctl processing function ip_process_ioctl() is called to extract the
12538  * arguments and process the ioctl.  To simplify extraction, ioctl commands
12539  * are "typed" based on the arguments they take (e.g., LIF_CMD which takes a
12540  * 'struct lifreq'), and a common extract function (e.g., ip_extract_lifreq())
12541  * is used to extract the ioctl's arguments.
12542  *
12543  * ip_process_ioctl determines if the ioctl needs to be serialized, and if
12544  * so goes thru the serialization primitive ipsq_try_enter. Then the
12545  * appropriate function to handle the ioctl is called based on the entry in
12546  * the ioctl table. ioctl completion is encapsulated in ip_ioctl_finish
12547  * which also refreleases the 'conn' that was refheld at the start of the
12548  * ioctl. Finally ipsq_exit is called if needed to exit the ipsq.
12549  *
12550  * Many exclusive ioctls go thru an internal down up sequence as part of
12551  * the operation. For example an attempt to change the IP address of an
12552  * ipif entails ipif_down, set address, ipif_up. Bringing down the interface
12553  * does all the cleanup such as deleting all ires that use this address.
12554  * Then we need to wait till all references to the interface go away.
12555  */
12556 void
12557 ip_process_ioctl(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *arg)
12558 {
12559         struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
12560         ip_ioctl_cmd_t *ipip = arg;
12561         ip_extract_func_t *extract_funcp;
12562         cmd_info_t ci;
12563         int err;
12564         boolean_t entered_ipsq = B_FALSE;

12566         ip3dbg(("ip_process_ioctl: ioctl %X\n", iocp->ioc_cmd));

12568         if (ipip == NULL)
12569                 ipip = ip_sioctl_lookup(iocp->ioc_cmd);

12571         /*
12572          * SIOCLIFADDIF needs to go thru a special path since the
12573          * ill may not exist yet. This happens in the case of lo0
12574          * which is created using this ioctl.
12575          */
12576         if (ipip->ipi_cmd == SIOCLIFADDIF) {
12577                 err = ip_sioctl_addif(NULL, NULL, q, mp, NULL, NULL);
12578                 DTRACE_PROBE4(ipif__ioctl, char *, "ip_process_ioctl finish",
12579                     int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12580                 ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12581                 return;
12582         }

12584         ci.ci_ipif = NULL;
12585         switch (ipip->ipi_cmd_type) {
12586         case MISC_CMD:
12587         case MSFILT_CMD:
12588                 /*
12589                  * All MISC_CMD ioctls come in here -- e.g. SIOCGLIFCONF.
12590                  */
12591                 if (ipip->ipi_cmd == IF_UNITSEL) {
12592                         /* ioctl comes down the ill */
12593                         ci.ci_ipif = ((ill_t *)q->q_ptr)->ill_ipif;
12594                         ipif_refhold(ci.ci_ipif);
12595                 }
12596                 err = 0;
12597                 ci.ci_sin = NULL;
12598                 ci.ci_sin6 = NULL;
12599                 ci.ci_lifr = NULL;
12600                 extract_funcp = NULL;
12601                 break;
```



```
12603         case IF_CMD:
12604         case LIF_CMD:
12605                 extract_funcp = ip_extract_lifreq;
12606                 break;

12608         case ARP_CMD:
12609         case XARP_CMD:
12610                 extract_funcp = ip_extract_arpreq;
12611                 break;

12613         default:
12614                 ASSERT(0);
12615         }

12617         if (extract_funcp != NULL) {
12618                 err = (*extract_funcp)(q, mp, ipip, &ci);
12619                 if (err != 0) {
12620                         DTRACE_PROBE4(ipif__ioctl,
12621                             char *, "ip_process_ioctl finish err",
12622                             int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12623                         ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12624                         return;
12625                 }

12627                 /*
12628                  * All of the extraction functions return a refheld ipif.
12629                  */
12630                 ASSERT(ci.ci_ipif != NULL);
12631         }

12633         if (!(ipip->ipi_flags & IPI_WR)) {
12634                 /*
12635                  * A return value of EINPROGRESS means the ioctl is
12636                  * either queued and waiting for some reason or has
12637                  * already completed.
12638                  */
12639                 err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip,
12640                     ci.ci_lifr);
12641                 if (ci.ci_ipif != NULL) {
12642                         DTRACE_PROBE4(ipif__ioctl,
12643                             char *, "ip_process_ioctl finish RD",
12644                             int, ipip->ipi_cmd, ill_t *, ci.ci_ipif->ipif_ill,
12645                             ipif_t *, ci.ci_ipif);
12646                         ipif_refrele(ci.ci_ipif);
12647                 } else {
12648                         DTRACE_PROBE4(ipif__ioctl,
12649                             char *, "ip_process_ioctl finish RD",
12650                             int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12651                 }
12652                 ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12653                 return;
12654         }

12656         ASSERT(ci.ci_ipif != NULL);

12658         /*
12659          * If ipsq is non-NULL, we are already being called exclusively
12660          */
12661         ASSERT(ipsq == NULL || IAM_WRITER_IPSQ(ipsq));
12662         if (ipsq == NULL) {
12663                 ipsq = ipsq_try_enter(ci.ci_ipif, NULL, q, mp, ip_process_ioctl,
12664                     NEW_OP, B_TRUE);
12665                 if (ipsq == NULL) {
12666                         ipif_refrele(ci.ci_ipif);
12667                         return;
```

```
12668                 }
12669                 entered_ipsq = B_TRUE;
12670         }
12671         /*
12672          * Release the ipif so that ipif_down and friends that wait for
12673          * references to go away are not misled about the current ipif_refcnt
12674          * values. We are writer so we can access the ipif even after releasing
12675          * the ipif.
12676          */
12677         ipif_refrele(ci.ci_ipif);

12679         ipsq_current_start(ipsq, ci.ci_ipif, ipip->ipi_cmd);

12681         /*
12682          * A return value of EINPROGRESS means the ioctl is
12683          * either queued and waiting for some reason or has
12684          * already completed.
12685          */
12686         err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip, ci.ci_lifr);

12688         DTRACE_PROBE4(ipif__ioctl, char *, "ip_process_ioctl finish WR",
12689             int, ipip->ipi_cmd,
12690             ill_t *, ci.ci_ipif == NULL ? NULL : ci.ci_ipif->ipif_ill,
12691             ipif_t *, ci.ci_ipif);
12692         ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), ipsq);

12694         if (entered_ipsq)
12695                 ipsq_exit(ipsq);
12696 }

12698 /*
12699  * Complete the ioctl. Typically ioctls use the mi package and need to
12700  * do mi_copyout/mi_copy_done.
12701  */
12702 void
12703 ip_ioctl_finish(queue_t *q, mblk_t *mp, int err, int mode, ipsq_t *ipsq)
12704 {
12705         conn_t  *connp = NULL;

12707         if (err == EINPROGRESS)
12708                 return;

12710         if (CONN_Q(q)) {
12711                 connp = Q_TO_CONN(q);
12712                 ASSERT(connp->conn_ref >= 2);
12713         }

12715         switch (mode) {
12716         case COPYOUT:
12717                 if (err == 0)
12718                         mi_copyout(q, mp);
12719                 else
12720                         mi_copy_done(q, mp, err);
12721                 break;

12723         case NO_COPYOUT:
12724                 mi_copy_done(q, mp, err);
12725                 break;

12727         default:
12728                 ASSERT(mode == CONN_CLOSE);     /* aborted through CONN_CLOSE */
12729                 break;
12730         }

12732         /*
12733          * The conn refhold and ioctlref placed on the conn at the start of the
```



```
12734          * ioctl are released here.
12735          */
12736         if (connp != NULL) {
12737                 CONN_DEC_IOCTLREF(connp);
12738                 CONN_OPER_PENDING_DONE(connp);
12739         }

12741         if (ipsq != NULL)
12742                 ipsq_current_finish(ipsq);
12743 }

12745 /* Handles all non data messages */
12746 void
12747 ip_wput_nondata(queue_t *q, mblk_t *mp)
12748 {
12749         mblk_t          *mp1;
12750         struct iocblk   *iocp;
12751         ip_ioctl_cmd_t  *ipip;
12752         conn_t          *connp;
12753         cred_t          *cr;
12754         char            *proto_str;

12756         if (CONN_Q(q))
12757                 connp = Q_TO_CONN(q);
12758         else
12759                 connp = NULL;

12761         switch (DB_TYPE(mp)) {
12762         case M_IOCTL:
12763                 /*
12764                  * IOCTL processing begins in ip_sioctl_copyin_setup which
12765                  * will arrange to copy in associated control structures.
12766                  */
12767                 ip_sioctl_copyin_setup(q, mp);
12768                 return;
12769         case M_IOCDATA:
12770                 /*
12771                  * Ensure that this is associated with one of our trans-
12772                  * parent ioctls.  If it's not ours, discard it if we're
12773                  * running as a driver, or pass it on if we're a module.
12774                  */
12775                 iocp = (struct iocblk *)mp->b_rptr;
12776                 ipip = ip_sioctl_lookup(iocp->ioc_cmd);
12777                 if (ipip == NULL) {
12778                         if (q->q_next == NULL) {
12779                                 goto nak;
12780                         } else {
12781                                 putnext(q, mp);
12782                         }
12783                         return;
12784                 }
12785                 if ((q->q_next != NULL) && !(ipip->ipi_flags & IPI_MODOK)) {
12786                         /*
12787                          * The ioctl is one we recognise, but is not consumed
12788                          * by IP as a module and we are a module, so we drop
12789                          */
12790                         goto nak;
12791                 }

12793                 /* IOCTL continuation following copyin or copyout. */
12794                 if (mi_copy_state(q, mp, NULL) == -1) {
12795                         /*
12796                          * The copy operation failed.  mi_copy_state already
12797                          * cleaned up, so we're out of here.
12798                          */
12799                         return;
```

```
12800                 }
12801                 /*
12802                  * If we just completed a copy in, we become writer and
12803                  * continue processing in ip_sioctl_copyin_done.  If it
12804                  * was a copy out, we call mi_copyout again.  If there is
12805                  * nothing more to copy out, it will complete the IOCTL.
12806                  */
12807                 if (MI_COPY_DIRECTION(mp) == MI_COPY_IN) {
12808                         if (!(mp1 = mp->b_cont) || !(mp1 = mp1->b_cont)) {
12809                                 mi_copy_done(q, mp, EPROTO);
12810                                 return;
12811                         }
12812                         /*
12813                          * Check for cases that need more copying.  A return
12814                          * value of 0 means a second copyin has been started,
12815                          * so we return; a return value of 1 means no more
12816                          * copying is needed, so we continue.
12817                          */
12818                         if (ipip->ipi_cmd_type == MSFILT_CMD &&
12819                             MI_COPY_COUNT(mp) == 1) {
12820                                 if (ip_copyin_msfilter(q, mp) == 0)
12821                                         return;
12822                         }
12823                         /*
12824                          * Refhold the conn, till the ioctl completes. This is
12825                          * needed in case the ioctl ends up in the pending mp
12826                          * list. Every mp in the ipx_pending_mp list must have
12827                          * a refhold on the conn to resume processing. The
12828                          * refhold is released when the ioctl completes
12829                          * (whether normally or abnormally). An ioctlref is also
12830                          * placed on the conn to prevent TCP from removing the
12831                          * queue needed to send the ioctl reply back.
12832                          * In all cases ip_ioctl_finish is called to finish
12833                          * the ioctl and release the refholds.
12834                          */
12835                         if (connp != NULL) {
12836                                 /* This is not a reentry */
12837                                 CONN_INC_REF(connp);
12838                                 CONN_INC_IOCTLREF(connp);
12839                         } else {
12840                                 if (!(ipip->ipi_flags & IPI_MODOK)) {
12841                                         mi_copy_done(q, mp, EINVAL);
12842                                         return;
12843                                 }
12844                         }

12846                         ip_process_ioctl(NULL, q, mp, ipip);

12848                 } else {
12849                         mi_copyout(q, mp);
12850                 }
12851                 return;

12853         case M_IOCNAK:
12854                 /*
12855                  * The only way we could get here is if a resolver didn't like
12856                  * an IOCTL we sent it.  This shouldn't happen.
12857                  */
12858                 (void) mi_strlog(q, 1, SL_ERROR|SL_TRACE,
12859                     "ip_wput_nondata: unexpected M_IOCNAK, ioc_cmd 0x%x",
12860                     ((struct iocblk *)mp->b_rptr)->ioc_cmd);
12861                 freemsg(mp);
12862                 return;
12863         case M_IOCACK:
12864                 /* /dev/ip shouldn't see this */
12865                 goto nak;
```



```
12866         case M_FLUSH:
12867                 if (*mp->b_rptr & FLUSHW)
12868                         flushq(q, FLUSHALL);
12869                 if (q->q_next) {
12870                         putnext(q, mp);
12871                         return;
12872                 }
12873                 if (*mp->b_rptr & FLUSHR) {
12874                         *mp->b_rptr &= ~FLUSHW;
12875                         qreply(q, mp);
12876                         return;
12877                 }
12878                 freemsg(mp);
12879                 return;
12880         case M_CTL:
12881                 break;
12882         case M_PROTO:
12883         case M_PCPROTO:
12884                 /*
12885                  * The only PROTO messages we expect are SNMP-related.
12886                  */
12887                 switch (((union T_primitives *)mp->b_rptr)->type) {
12888                 case T_SVR4_OPTMGMT_REQ:
12889                         ip2dbg(("ip_wput_nondata: T_SVR4_OPTMGMT_REQ "
12890                             "flags %x\n",
12891                             ((struct T_optmgmt_req *)mp->b_rptr)->MGMT_flags));

12893                         if (connp == NULL) {
12894                                 proto_str = "T_SVR4_OPTMGMT_REQ";
12895                                 goto protonak;
12896                         }

12898                         /*
12899                          * All Solaris components should pass a db_credp
12900                          * for this TPI message, hence we ASSERT.
12901                          * But in case there is some other M_PROTO that looks
12902                          * like a TPI message sent by some other kernel
12903                          * component, we check and return an error.
12904                          */
12905                         cr = msg_getcred(mp, NULL);
12906                         ASSERT(cr != NULL);
12907                         if (cr == NULL) {
12908                                 mp = mi_tpi_err_ack_alloc(mp, TSYSERR, EINVAL);
12909                                 if (mp != NULL)
12910                                         qreply(q, mp);
12911                                 return;
12912                         }

12914                         if (!snmpcom_req(q, mp, ip_snmp_set, ip_snmp_get, cr)) {
12915                                 proto_str = "Bad SNMPCOM request?";
12916                                 goto protonak;
12917                         }
12918                         return;
12919                 default:
12920                         ip1dbg(("ip_wput_nondata: dropping M_PROTO prim %u\n",
12921                             (int)*(uint_t *)mp->b_rptr));
12922                         freemsg(mp);
12923                         return;
12924                 }
12925         default:
12926                 break;
12927         }
12928         if (q->q_next) {
12929                 putnext(q, mp);
12930         } else
12931                 freemsg(mp);
```

```
12932         return;

12934 nak:
12935         iocp->ioc_error = EINVAL;
12936         mp->b_datap->db_type = M_IOCNAK;
12937         iocp->ioc_count = 0;
12938         qreply(q, mp);
12939         return;

12941 protonak:
12942         cmn_err(CE_NOTE, "IP doesn't process %s as a module", proto_str);
12943         if ((mp = mi_tpi_err_ack_alloc(mp, TPROTO, EINVAL)) != NULL)
12944                 qreply(q, mp);
12945 }

12947 /*
12948  * Process IP options in an outbound packet.  Verify that the nexthop in a
12949  * strict source route is onlink.
12950  * Returns non-zero if something fails in which case an ICMP error has been
12951  * sent and mp freed.
12952  *
12953  * Assumes the ULP has called ip_massage_options to move nexthop into ipha_dst.
12954  */
12955 int
12956 ip_output_options(mblk_t *mp, ipha_t *ipha, ip_xmit_attr_t *ixa, ill_t *ill)
12957 {
12958         ipoptp_t        opts;
12959         uchar_t         *opt;
12960         uint8_t         optval;
12961         uint8_t         optlen;
12962         ipaddr_t        dst;
12963         intptr_t        code = 0;
12964         ire_t           *ire;
12965         ip_stack_t      *ipst = ixa->ixa_ipst;
12966         ip_recv_attr_t  iras;

12968         ip2dbg(("ip_output_options\n"));

12970         dst = ipha->ipha_dst;
12971         for (optval = ipoptp_first(&opts, ipha);
12972             optval != IPOPT_EOL;
12973             optval = ipoptp_next(&opts)) {
12974                 opt = opts.ipoptp_cur;
12975                 optlen = opts.ipoptp_len;
12976                 ip2dbg(("ip_output_options: opt %d, len %d\n",
12977                     optval, optlen));
12978                 switch (optval) {
12979                         uint32_t off;
12980                 case IPOPT_SSRR:
12981                 case IPOPT_LSRR:
12982                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
12983                                 ip1dbg((
12984                                     "ip_output_options: bad option offset\n"));
12985                                 code = (char *)&opt[IPOPT_OLEN] -
12986                                     (char *)ipha;
12987                                 goto param_prob;
12988                         }
12989                         off = opt[IPOPT_OFFSET];
12990                         ip1dbg(("ip_output_options: next hop 0x%x\n",
12991                             ntohl(dst)));
12992                         /*
12993                          * For strict: verify that dst is directly
12994                          * reachable.
12995                          */
12996                         if (optval == IPOPT_SSRR) {
12997                                 ire = ire_ftable_lookup_v4(dst, 0, 0,
```



```
12998                                     IRE_INTERFACE, NULL, ALL_ZONES,
12999                                     ixa->ixa_tsl,
13000                                     MATCH_IRE_TYPE | MATCH_IRE_SECATTR, 0, ipst,
13001                                     NULL);
13002                                 if (ire == NULL) {
13003                                         ip1dbg(("ip_output_options: SSRR not"
13004                                             " directly reachable: 0x%x\n",
13005                                             ntohl(dst)));
13006                                         goto bad_src_route;
13007                                 }
13008                                 ire_refrele(ire);
13009                         }
13010                         break;
13011                 case IPOPT_RR:
13012                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
13013                                 ip1dbg((
13014                                     "ip_output_options: bad option offset\n"));
13015                                 code = (char *)&opt[IPOPT_OLEN] -
13016                                     (char *)ipha;
13017                                 goto param_prob;
13018                         }
13019                         break;
13020                 case IPOPT_TS:
13021                         /*
13022                          * Verify that length >=5 and that there is either
13023                          * room for another timestamp or that the overflow
13024                          * counter is not maxed out.
13025                          */
13026                         code = (char *)&opt[IPOPT_OLEN] - (char *)ipha;
13027                         if (optlen < IPOPT_MINLEN_IT) {
13028                                 goto param_prob;
13029                         }
13030                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
13031                                 ip1dbg((
13032                                     "ip_output_options: bad option offset\n"));
13033                                 code = (char *)&opt[IPOPT_OFFSET] -
13034                                     (char *)ipha;
13035                                 goto param_prob;
13036                         }
13037                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
13038                         case IPOPT_TS_TSONLY:
13039                                 off = IPOPT_TS_TIMELEN;
13040                                 break;
13041                         case IPOPT_TS_TSANDADDR:
13042                         case IPOPT_TS_PRESPEC:
13043                         case IPOPT_TS_PRESPEC_RFC791:
13044                                 off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
13045                                 break;
13046                         default:
13047                                 code = (char *)&opt[IPOPT_POS_OV_FLG] -
13048                                     (char *)ipha;
13049                                 goto param_prob;
13050                         }
13051                         if (opt[IPOPT_OFFSET] - 1 + off > optlen &&
13052                             (opt[IPOPT_POS_OV_FLG] & 0xF0) == 0xF0) {
13053                                 /*
13054                                  * No room and the overflow counter is 15
13055                                  * already.
13056                                  */
13057                                 goto param_prob;
13058                         }
13059                         break;
13060                 }
13061         }

13063         if ((opts.ipoptp_flags & IPOPTP_ERROR) == 0)
```

```
13064                 return (0);

13066         ip1dbg(("ip_output_options: error processing IP options."));
13067         code = (char *)&opt[IPOPT_OFFSET] - (char *)ipha;

13069 param_prob:
13070         bzero(&iras, sizeof (iras));
13071         iras.ira_ill = iras.ira_rill = ill;
13072         iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
13073         iras.ira_rifindex = iras.ira_ruifindex;
13074         iras.ira_flags = IRAF_IS_IPV4;

13076         ip_drop_output("ip_output_options", mp, ill);
13077         icmp_param_problem(mp, (uint8_t)code, &iras);
13078         ASSERT(!(iras.ira_flags & IRAF_IPSEC_SECURE));
13079         return (-1);

13081 bad_src_route:
13082         bzero(&iras, sizeof (iras));
13083         iras.ira_ill = iras.ira_rill = ill;
13084         iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
13085         iras.ira_rifindex = iras.ira_ruifindex;
13086         iras.ira_flags = IRAF_IS_IPV4;

13088         ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
13089         icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, &iras);
13090         ASSERT(!(iras.ira_flags & IRAF_IPSEC_SECURE));
13091         return (-1);
13092 }

13094 /*
13095  * The maximum value of conn_drain_list_cnt is CONN_MAXDRAINCNT.
13096  * conn_drain_list_cnt can be changed by setting conn_drain_nthreads
13097  * thru /etc/system.
13098  */
13099 #define CONN_MAXDRAINCNT        64

13101 static void
13102 conn_drain_init(ip_stack_t *ipst)
13103 {
13104         int i, j;
13105         idl_tx_list_t *itl_tx;

13107         ipst->ips_conn_drain_list_cnt = conn_drain_nthreads;

13109         if ((ipst->ips_conn_drain_list_cnt == 0) ||
13110             (ipst->ips_conn_drain_list_cnt > CONN_MAXDRAINCNT)) {
13111                 /*
13112                  * Default value of the number of drainers is the
13113                  * number of cpus, subject to maximum of 8 drainers.
13114                  */
13115                 if (boot_max_ncpus != -1)
13116                         ipst->ips_conn_drain_list_cnt = MIN(boot_max_ncpus, 8);
13117                 else
13118                         ipst->ips_conn_drain_list_cnt = MIN(max_ncpus, 8);
13119         }

13121         ipst->ips_idl_tx_list =
13122             kmem_zalloc(TX_FANOUT_SIZE * sizeof (idl_tx_list_t), KM_SLEEP);
13123         for (i = 0; i < TX_FANOUT_SIZE; i++) {
13124                 itl_tx =  &ipst->ips_idl_tx_list[i];
13125                 itl_tx->txl_drain_list =
13126                     kmem_zalloc(ipst->ips_conn_drain_list_cnt *
13127                     sizeof (idl_t), KM_SLEEP);
13128                 mutex_init(&itl_tx->txl_lock, NULL, MUTEX_DEFAULT, NULL);
13129                 for (j = 0; j < ipst->ips_conn_drain_list_cnt; j++) {
```



```
13130                         mutex_init(&itl_tx->txl_drain_list[j].idl_lock, NULL,
13131                             MUTEX_DEFAULT, NULL);
13132                         itl_tx->txl_drain_list[j].idl_itl = itl_tx;
13133                 }
13134         }
13135 }

13137 static void
13138 conn_drain_fini(ip_stack_t *ipst)
13139 {
13140         int i;
13141         idl_tx_list_t *itl_tx;

13143         for (i = 0; i < TX_FANOUT_SIZE; i++) {
13144                 itl_tx =  &ipst->ips_idl_tx_list[i];
13145                 kmem_free(itl_tx->txl_drain_list,
13146                     ipst->ips_conn_drain_list_cnt * sizeof (idl_t));
13147         }
13148         kmem_free(ipst->ips_idl_tx_list,
13149             TX_FANOUT_SIZE * sizeof (idl_tx_list_t));
13150         ipst->ips_idl_tx_list = NULL;
13151 }

13153 /*
13154  * Flow control has blocked us from proceeding.  Insert the given conn in one
13155  * of the conn drain lists.  When flow control is unblocked, either ip_wsrv()
13156  * (STREAMS) or ill_flow_enable() (direct) will be called back, which in turn
13157  * will call conn_walk_drain().  See the flow control notes at the top of this
13158  * file for more details.
13159  */
13160 void
13161 conn_drain_insert(conn_t *connp, idl_tx_list_t *tx_list)
13162 {
13163         idl_t   *idl = tx_list->txl_drain_list;
13164         uint_t  index;
13165         ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;

13167         mutex_enter(&connp->conn_lock);
13168         if (connp->conn_state_flags & CONN_CLOSING) {
13169                 /*
13170                  * The conn is closing as a result of which CONN_CLOSING
13171                  * is set. Return.
13172                  */
13173                 mutex_exit(&connp->conn_lock);
13174                 return;
13175         } else if (connp->conn_idl == NULL) {
13176                 /*
13177                  * Assign the next drain list round robin. We dont' use
13178                  * a lock, and thus it may not be strictly round robin.
13179                  * Atomicity of load/stores is enough to make sure that
13180                  * conn_drain_list_index is always within bounds.
13181                  */
13182                 index = tx_list->txl_drain_index;
13183                 ASSERT(index < ipst->ips_conn_drain_list_cnt);
13184                 connp->conn_idl = &tx_list->txl_drain_list[index];
13185                 index++;
13186                 if (index == ipst->ips_conn_drain_list_cnt)
13187                         index = 0;
13188                 tx_list->txl_drain_index = index;
13189         } else {
13190                 ASSERT(connp->conn_idl->idl_itl == tx_list);
13191         }
13192         mutex_exit(&connp->conn_lock);

13194         idl = connp->conn_idl;
13195         mutex_enter(&idl->idl_lock);
```

```
13196         if ((connp->conn_drain_prev != NULL) ||
13197             (connp->conn_state_flags & CONN_CLOSING)) {
13198                 /*
13199                  * The conn is either already in the drain list or closing.
13200                  * (We needed to check for CONN_CLOSING again since close can
13201                  * sneak in between dropping conn_lock and acquiring idl_lock.)
13202                  */
13203                 mutex_exit(&idl->idl_lock);
13204                 return;
13205         }

13207         /*
13208          * The conn is not in the drain list. Insert it at the
13209          * tail of the drain list. The drain list is circular
13210          * and doubly linked. idl_conn points to the 1st element
13211          * in the list.
13212          */
13213         if (idl->idl_conn == NULL) {
13214                 idl->idl_conn = connp;
13215                 connp->conn_drain_next = connp;
13216                 connp->conn_drain_prev = connp;
13217         } else {
13218                 conn_t *head = idl->idl_conn;

13220                 connp->conn_drain_next = head;
13221                 connp->conn_drain_prev = head->conn_drain_prev;
13222                 head->conn_drain_prev->conn_drain_next = connp;
13223                 head->conn_drain_prev = connp;
13224         }
13225         /*
13226          * For non streams based sockets assert flow control.
13227          */
13228         conn_setqfull(connp, NULL);
13229         mutex_exit(&idl->idl_lock);
13230 }

13232 static void
13233 conn_drain_remove(conn_t *connp)
13234 {
13235         idl_t *idl = connp->conn_idl;

13237         if (idl != NULL) {
13238                 /*
13239                  * Remove ourself from the drain list.
13240                  */
13241                 if (connp->conn_drain_next == connp) {
13242                         /* Singleton in the list */
13243                         ASSERT(connp->conn_drain_prev == connp);
13244                         idl->idl_conn = NULL;
13245                 } else {
13246                         connp->conn_drain_prev->conn_drain_next =
13247                             connp->conn_drain_next;
13248                         connp->conn_drain_next->conn_drain_prev =
13249                             connp->conn_drain_prev;
13250                         if (idl->idl_conn == connp)
13251                                 idl->idl_conn = connp->conn_drain_next;
13252                 }

13254                 /*
13255                  * NOTE: because conn_idl is associated with a specific drain
13256                  * list which in turn is tied to the index the TX ring
13257                  * (txl_cookie) hashes to, and because the TX ring can change
13258                  * over the lifetime of the conn_t, we must clear conn_idl so
13259                  * a subsequent conn_drain_insert() will set conn_idl again
13260                  * based on the latest txl_cookie.
13261                  */
```



```
13262                 connp->conn_idl = NULL;
13263         }
13264         connp->conn_drain_next = NULL;
13265         connp->conn_drain_prev = NULL;

13267         conn_clrqfull(connp, NULL);
13268         /*
13269          * For streams based sockets open up flow control.
13270          */
13271         if (!IPCL_IS_NONSTR(connp))
13272                 enableok(connp->conn_wq);
13273 }

13275 /*
13276  * This conn is closing, and we are called from ip_close. OR
13277  * this conn is draining because flow-control on the ill has been relieved.
13278  *
13279  * We must also need to remove conn's on this idl from the list, and also
13280  * inform the sockfs upcalls about the change in flow-control.
13281  */
13282 static void
13283 conn_drain(conn_t *connp, boolean_t closing)
13284 {
13285         idl_t *idl;
13286         conn_t *next_connp;

13288         /*
13289          * connp->conn_idl is stable at this point, and no lock is needed
13290          * to check it. If we are called from ip_close, close has already
13291          * set CONN_CLOSING, thus freezing the value of conn_idl, and
13292          * called us only because conn_idl is non-null. If we are called thru
13293          * service, conn_idl could be null, but it cannot change because
13294          * service is single-threaded per queue, and there cannot be another
13295          * instance of service trying to call conn_drain_insert on this conn
13296          * now.
13297          */
13298         ASSERT(!closing || connp == NULL || connp->conn_idl != NULL);

13300         /*
13301          * If the conn doesn't exist or is not on a drain list, bail.
13302          */
13303         if (connp == NULL || connp->conn_idl == NULL ||
13304             connp->conn_drain_prev == NULL) {
13305                 return;
13306         }

13308         idl = connp->conn_idl;
13309         ASSERT(MUTEX_HELD(&idl->idl_lock));

13311         if (!closing) {
13312                 next_connp = connp->conn_drain_next;
13313                 while (next_connp != connp) {
13314                         conn_t *delconnp = next_connp;

13316                         next_connp = next_connp->conn_drain_next;
13317                         conn_drain_remove(delconnp);
13318                 }
13319                 ASSERT(connp->conn_drain_next == idl->idl_conn);
13320         }
13321         conn_drain_remove(connp);
13322 }

13324 /*
13325  * Write service routine. Shared perimeter entry point.
13326  * The device queue's messages has fallen below the low water mark and STREAMS
13327  * has backenabled the ill_wq. Send sockfs notification about flow-control on
```

```
13328  * each waiting conn.
13329  */
13330 void
13331 ip_wsrv(queue_t *q)
13332 {
13333         ill_t   *ill;

13335         ill = (ill_t *)q->q_ptr;
13336         if (ill->ill_state_flags == 0) {
13337                 ip_stack_t *ipst = ill->ill_ipst;

13339                 /*
13340                  * The device flow control has opened up.
13341                  * Walk through conn drain lists and qenable the
13342                  * first conn in each list. This makes sense only
13343                  * if the stream is fully plumbed and setup.
13344                  * Hence the ill_state_flags check above.
13345                  */
13346                 ip1dbg(("ip_wsrv: walking\n"));
13347                 conn_walk_drain(ipst, &ipst->ips_idl_tx_list[0]);
13348                 enableok(ill->ill_wq);
13349         }
13350 }

13352 /*
13353  * Callback to disable flow control in IP.
13354  *
13355  * This is a mac client callback added when the DLD_CAPAB_DIRECT capability
13356  * is enabled.
13357  *
13358  * When MAC_TX() is not able to send any more packets, dld sets its queue
13359  * to QFULL and enable the STREAMS flow control. Later, when the underlying
13360  * driver is able to continue to send packets, it calls mac_tx_(ring_)update()
13361  * function and wakes up corresponding mac worker threads, which in turn
13362  * calls this callback function, and disables flow control.
13363  */
13364 void
13365 ill_flow_enable(void *arg, ip_mac_tx_cookie_t cookie)
13366 {
13367         ill_t *ill = (ill_t *)arg;
13368         ip_stack_t *ipst = ill->ill_ipst;
13369         idl_tx_list_t *idl_txl;

13371         idl_txl = &ipst->ips_idl_tx_list[IDLHASHINDEX(cookie)];
13372         mutex_enter(&idl_txl->txl_lock);
13373         /* add code to to set a flag to indicate idl_txl is enabled */
13374         conn_walk_drain(ipst, idl_txl);
13375         mutex_exit(&idl_txl->txl_lock);
13376 }

13378 /*
13379  * Flow control has been relieved and STREAMS has backenabled us; drain
13380  * all the conn lists on `tx_list'.
13381  */
13382 static void
13383 conn_walk_drain(ip_stack_t *ipst, idl_tx_list_t *tx_list)
13384 {
13385         int i;
13386         idl_t *idl;

13388         IP_STAT(ipst, ip_conn_walk_drain);

13390         for (i = 0; i < ipst->ips_conn_drain_list_cnt; i++) {
13391                 idl = &tx_list->txl_drain_list[i];
13392                 mutex_enter(&idl->idl_lock);
13393                 conn_drain(idl->idl_conn, B_FALSE);
```



```
13394                 mutex_exit(&idl->idl_lock);
13395         }
13396 }

13398 /*
13399  * Determine if the ill and multicast aspects of that packets
13400  * "matches" the conn.
13401  */
13402 boolean_t
13403 conn_wantpacket(conn_t *connp, ip_recv_attr_t *ira, ipha_t *ipha)
13404 {
13405         ill_t           *ill = ira->ira_rill;
13406         zoneid_t        zoneid = ira->ira_zoneid;
13407         uint_t          in_ifindex;
13408         ipaddr_t        dst, src;

13410         dst = ipha->ipha_dst;
13411         src = ipha->ipha_src;

13413         /*
13414          * conn_incoming_ifindex is set by IP_BOUND_IF which limits
13415          * unicast, broadcast and multicast reception to
13416          * conn_incoming_ifindex.
13417          * conn_wantpacket is called for unicast, broadcast and
13418          * multicast packets.
13419          */
13420         in_ifindex = connp->conn_incoming_ifindex;

13422         /* mpathd can bind to the under IPMP interface, which we allow */
13423         if (in_ifindex != 0 && in_ifindex != ill->ill_phyint->phyint_ifindex) {
13424                 if (!IS_UNDER_IPMP(ill))
13425                         return (B_FALSE);

13427                 if (in_ifindex != ipmp_ill_get_ipmp_ifindex(ill))
13428                         return (B_FALSE);
13429         }

13431         if (!IPCL_ZONE_MATCH(connp, zoneid))
13432                 return (B_FALSE);

13434         if (!(ira->ira_flags & IRAF_MULTICAST))
13435                 return (B_TRUE);

13437         if (connp->conn_multi_router) {
13438                 /* multicast packet and multicast router socket: send up */
13439                 return (B_TRUE);
13440         }

13442         if (ipha->ipha_protocol == IPPROTO_PIM ||
13443             ipha->ipha_protocol == IPPROTO_RSVP)
13444                 return (B_TRUE);

13446         return (conn_hasmembers_ill_withsrc_v4(connp, dst, src, ira->ira_ill));
13447 }

13449 void
13450 conn_setqfull(conn_t *connp, boolean_t *flow_stopped)
13451 {
13452         if (IPCL_IS_NONSTR(connp)) {
13453                 (*connp->conn_upcalls->su_txq_full)
13454                     (connp->conn_upper_handle, B_TRUE);
13455                 if (flow_stopped != NULL)
13456                         *flow_stopped = B_TRUE;
13457         } else {
13458                 queue_t *q = connp->conn_wq;
```

```
13460                 ASSERT(q != NULL);
13461                 if (!(q->q_flag & QFULL)) {
13462                         mutex_enter(QLOCK(q));
13463                         if (!(q->q_flag & QFULL)) {
13464                                 /* still need to set QFULL */
13465                                 q->q_flag |= QFULL;
13466                                 /* set flow_stopped to true under QLOCK */
13467                                 if (flow_stopped != NULL)
13468                                         *flow_stopped = B_TRUE;
13469                                 mutex_exit(QLOCK(q));
13470                         } else {
13471                                 /* flow_stopped is left unchanged */
13472                                 mutex_exit(QLOCK(q));
13473                         }
13474                 }
13475         }
13476 }

13478 void
13479 conn_clrqfull(conn_t *connp, boolean_t *flow_stopped)
13480 {
13481         if (IPCL_IS_NONSTR(connp)) {
13482                 (*connp->conn_upcalls->su_txq_full)
13483                     (connp->conn_upper_handle, B_FALSE);
13484                 if (flow_stopped != NULL)
13485                         *flow_stopped = B_FALSE;
13486         } else {
13487                 queue_t *q = connp->conn_wq;

13489                 ASSERT(q != NULL);
13490                 if (q->q_flag & QFULL) {
13491                         mutex_enter(QLOCK(q));
13492                         if (q->q_flag & QFULL) {
13493                                 q->q_flag &= ~QFULL;
13494                                 /* set flow_stopped to false under QLOCK */
13495                                 if (flow_stopped != NULL)
13496                                         *flow_stopped = B_FALSE;
13497                                 mutex_exit(QLOCK(q));
13498                                 if (q->q_flag & QWANTW)
13499                                         qbackenable(q, 0);
13500                         } else {
13501                                 /* flow_stopped is left unchanged */
13502                                 mutex_exit(QLOCK(q));
13503                         }
13504                 }
13505         }

13507         mutex_enter(&connp->conn_lock);
13508         connp->conn_blocked = B_FALSE;
13509         mutex_exit(&connp->conn_lock);
13510 }

13512 /*
13513  * Return the length in bytes of the IPv4 headers (base header, label, and
13514  * other IP options) that will be needed based on the
13515  * ip_pkt_t structure passed by the caller.
13516  *
13517  * The returned length does not include the length of the upper level
13518  * protocol (ULP) header.
13519  * The caller needs to check that the length doesn't exceed the max for IPv4.
13520  */
13521 int
13522 ip_total_hdrs_len_v4(const ip_pkt_t *ipp)
13523 {
13524         int len;
```



```
13526         len = IP_SIMPLE_HDR_LENGTH;
13527         if (ipp->ipp_fields & IPPF_LABEL_V4) {
13528                 ASSERT(ipp->ipp_label_len_v4 != 0);
13529                 /* We need to round up here */
13530                 len += (ipp->ipp_label_len_v4 + 3) & ~3;
13531         }

13533         if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
13534                 ASSERT(ipp->ipp_ipv4_options_len != 0);
13535                 ASSERT((ipp->ipp_ipv4_options_len & 3) == 0);
13536                 len += ipp->ipp_ipv4_options_len;
13537         }
13538         return (len);
13539 }

13541 /*
13542  * All-purpose routine to build an IPv4 header with options based
13543  * on the abstract ip_pkt_t.
13544  *
13545  * The caller has to set the source and destination address as well as
13546  * ipha_length. The caller has to massage any source route and compensate
13547  * for the ULP pseudo-header checksum due to the source route.
13548  */
13549 void
13550 ip_build_hdrs_v4(uchar_t *buf, uint_t buf_len, const ip_pkt_t *ipp,
13551     uint8_t protocol)
13552 {
13553         ipha_t  *ipha = (ipha_t *)buf;
13554         uint8_t *cp;

13556         /* Initialize IPv4 header */
13557         ipha->ipha_type_of_service = ipp->ipp_type_of_service;
13558         ipha->ipha_length = 0;  /* Caller will set later */
13559         ipha->ipha_ident = 0;
13560         ipha->ipha_fragment_offset_and_flags = 0;
13561         ipha->ipha_ttl = ipp->ipp_unicast_hops;
13562         ipha->ipha_protocol = protocol;
13563         ipha->ipha_hdr_checksum = 0;

13565         if ((ipp->ipp_fields & IPPF_ADDR) &&
13566             IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr))
13567                 ipha->ipha_src = ipp->ipp_addr_v4;

13569         cp = (uint8_t *)&ipha[1];
13570         if (ipp->ipp_fields & IPPF_LABEL_V4) {
13571                 ASSERT(ipp->ipp_label_len_v4 != 0);
13572                 bcopy(ipp->ipp_label_v4, cp, ipp->ipp_label_len_v4);
13573                 cp += ipp->ipp_label_len_v4;
13574                 /* We need to round up here */
13575                 while ((uintptr_t)cp & 0x3) {
13576                         *cp++ = IPOPT_NOP;
13577                 }
13578         }

13580         if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
13581                 ASSERT(ipp->ipp_ipv4_options_len != 0);
13582                 ASSERT((ipp->ipp_ipv4_options_len & 3) == 0);
13583                 bcopy(ipp->ipp_ipv4_options, cp, ipp->ipp_ipv4_options_len);
13584                 cp += ipp->ipp_ipv4_options_len;
13585         }
13586         ipha->ipha_version_and_hdr_length =
13587             (uint8_t)((IP_VERSION << 4) + buf_len / 4);

13589         ASSERT((int)(cp - buf) == buf_len);
13590 }
```

```
13592 /* Allocate the private structure */
13593 static int
13594 ip_priv_alloc(void **bufp)
13595 {
13596         void    *buf;

13598         if ((buf = kmem_alloc(sizeof (ip_priv_t), KM_NOSLEEP)) == NULL)
13599                 return (ENOMEM);

13601         *bufp = buf;
13602         return (0);
13603 }

13605 /* Function to delete the private structure */
13606 void
13607 ip_priv_free(void *buf)
13608 {
13609         ASSERT(buf != NULL);
13610         kmem_free(buf, sizeof (ip_priv_t));
13611 }

13613 /*
13614  * The entry point for IPPF processing.
13615  * If the classifier (IPGPC_CLASSIFY) is not loaded and configured, the
13616  * routine just returns.
13617  *
13618  * When called, ip_process generates an ipp_packet_t structure
13619  * which holds the state information for this packet and invokes the
13620  * the classifier (via ipp_packet_process). The classification, depending on
13621  * configured filters, results in a list of actions for this packet. Invoking
13622  * an action may cause the packet to be dropped, in which case we return NULL.
13623  * proc indicates the callout position for
13624  * this packet and ill is the interface this packet arrived on or will leave
13625  * on (inbound and outbound resp.).
13626  *
13627  * We do the processing on the rill (mapped to the upper if ipmp), but MIB
13628  * on the ill corrsponding to the destination IP address.
13629  */
13630 mblk_t *
13631 ip_process(ip_proc_t proc, mblk_t *mp, ill_t *rill, ill_t *ill)
13632 {
13633         ip_priv_t       *priv;
13634         ipp_action_id_t aid;
13635         int             rc = 0;
13636         ipp_packet_t    *pp;

13638         /* If the classifier is not loaded, return  */
13639         if ((aid = ipp_action_lookup(IPGPC_CLASSIFY)) == IPP_ACTION_INVAL) {
13640                 return (mp);
13641         }

13643         ASSERT(mp != NULL);

13645         /* Allocate the packet structure */
13646         rc = ipp_packet_alloc(&pp, "ip", aid);
13647         if (rc != 0)
13648                 goto drop;

13650         /* Allocate the private structure */
13651         rc = ip_priv_alloc((void **)&priv);
13652         if (rc != 0) {
13653                 ipp_packet_free(pp);
13654                 goto drop;
13655         }
13656         priv->proc = proc;
13657         priv->ill_index = ill_get_upper_ifindex(rill);
```



```
13659         ipp_packet_set_private(pp, priv, ip_priv_free);
13660         ipp_packet_set_data(pp, mp);

13662         /* Invoke the classifier */
13663         rc = ipp_packet_process(&pp);
13664         if (pp != NULL) {
13665                 mp = ipp_packet_get_data(pp);
13666                 ipp_packet_free(pp);
13667                 if (rc != 0)
13668                         goto drop;
13669                 return (mp);
13670         } else {
13671                 /* No mp to trace in ip_drop_input/ip_drop_output  */
13672                 mp = NULL;
13673         }
13674 drop:
13675         if (proc == IPP_LOCAL_IN || proc == IPP_FWD_IN) {
13676                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
13677                 ip_drop_input("ip_process", mp, ill);
13678         } else {
13679                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
13680                 ip_drop_output("ip_process", mp, ill);
13681         }
13682         freemsg(mp);
13683         return (NULL);
13684 }

13686 /*
13687  * Propagate a multicast group membership operation (add/drop) on
13688  * all the interfaces crossed by the related multirt routes.
13689  * The call is considered successful if the operation succeeds
13690  * on at least one interface.
13691  *
13692  * This assumes that a set of IRE_HOST/RTF_MULTIRT has been created for the
13693  * multicast addresses with the ire argument being the first one.
13694  * We walk the bucket to find all the of those.
13695  *
13696  * Common to IPv4 and IPv6.
13697  */
13698 static int
13699 ip_multirt_apply_membership(int (*fn)(conn_t *, boolean_t,
13700     const in6_addr_t *, ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *),
13701     ire_t *ire, conn_t *connp, boolean_t checkonly, const in6_addr_t *v6group,
13702     mcast_record_t fmode, const in6_addr_t *v6src)
13703 {
13704         ire_t           *ire_gw;
13705         irb_t           *irb;
13706         int             ifindex;
13707         int             error = 0;
13708         int             result;
13709         ip_stack_t      *ipst = ire->ire_ipst;
13710         ipaddr_t        group;
13711         boolean_t       isv6;
13712         int             match_flags;

13714         if (IN6_IS_ADDR_V4MAPPED(v6group)) {
13715                 IN6_V4MAPPED_TO_IPADDR(v6group, group);
13716                 isv6 = B_FALSE;
13717         } else {
13718                 isv6 = B_TRUE;
13719         }

13721         irb = ire->ire_bucket;
13722         ASSERT(irb != NULL);
```

```
13724         result = 0;
13725         irb_refhold(irb);
13726         for (; ire != NULL; ire = ire->ire_next) {
13727                 if ((ire->ire_flags & RTF_MULTIRT) == 0)
13728                         continue;

13730                 /* We handle -ifp routes by matching on the ill if set */
13731                 match_flags = MATCH_IRE_TYPE;
13732                 if (ire->ire_ill != NULL)
13733                         match_flags |= MATCH_IRE_ILL;

13735                 if (isv6) {
13736                         if (!IN6_ARE_ADDR_EQUAL(&ire->ire_addr_v6, v6group))
13737                                 continue;

13739                         ire_gw = ire_ftable_lookup_v6(&ire->ire_gateway_addr_v6,
13740                             0, 0, IRE_INTERFACE, ire->ire_ill, ALL_ZONES, NULL,
13741                             match_flags, 0, ipst, NULL);
13742                 } else {
13743                         if (ire->ire_addr != group)
13744                                 continue;

13746                         ire_gw = ire_ftable_lookup_v4(ire->ire_gateway_addr,
13747                             0, 0, IRE_INTERFACE, ire->ire_ill, ALL_ZONES, NULL,
13748                             match_flags, 0, ipst, NULL);
13749                 }
13750                 /* No interface route exists for the gateway; skip this ire. */
13751                 if (ire_gw == NULL)
13752                         continue;
13753                 if (ire_gw->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) {
13754                         ire_refrele(ire_gw);
13755                         continue;
13756                 }
13757                 ASSERT(ire_gw->ire_ill != NULL);        /* IRE_INTERFACE */
13758                 ifindex = ire_gw->ire_ill->ill_phyint->phyint_ifindex;

13760                 /*
13761                  * The operation is considered a success if
13762                  * it succeeds at least once on any one interface.
13763                  */
13764                 error = fn(connp, checkonly, v6group, INADDR_ANY, ifindex,
13765                     fmode, v6src);
13766                 if (error == 0)
13767                         result = CGTP_MCAST_SUCCESS;

13769                 ire_refrele(ire_gw);
13770         }
13771         irb_refrele(irb);
13772         /*
13773          * Consider the call as successful if we succeeded on at least
13774          * one interface. Otherwise, return the last encountered error.
13775          */
13776         return (result == CGTP_MCAST_SUCCESS ? 0 : error);
13777 }

13779 /*
13780  * Return the expected CGTP hooks version number.
13781  */
13782 int
13783 ip_cgtp_filter_supported(void)
13784 {
13785         return (ip_cgtp_filter_rev);
13786 }

13788 /*
13789  * CGTP hooks can be registered by invoking this function.
```



```
13790  * Checks that the version number matches.
13791  */
13792 int
13793 ip_cgtp_filter_register(netstackid_t stackid, cgtp_filter_ops_t *ops)
13794 {
13795         netstack_t *ns;
13796         ip_stack_t *ipst;

13798         if (ops->cfo_filter_rev != CGTP_FILTER_REV)
13799                 return (ENOTSUP);

13801         ns = netstack_find_by_stackid(stackid);
13802         if (ns == NULL)
13803                 return (EINVAL);
13804         ipst = ns->netstack_ip;
13805         ASSERT(ipst != NULL);

13807         if (ipst->ips_ip_cgtp_filter_ops != NULL) {
13808                 netstack_rele(ns);
13809                 return (EALREADY);
13810         }

13812         ipst->ips_ip_cgtp_filter_ops = ops;

13814         ill_set_inputfn_all(ipst);

13816         netstack_rele(ns);
13817         return (0);
13818 }

13820 /*
13821  * CGTP hooks can be unregistered by invoking this function.
13822  * Returns ENXIO if there was no registration.
13823  * Returns EBUSY if the ndd variable has not been turned off.
13824  */
13825 int
13826 ip_cgtp_filter_unregister(netstackid_t stackid)
13827 {
13828         netstack_t *ns;
13829         ip_stack_t *ipst;

13831         ns = netstack_find_by_stackid(stackid);
13832         if (ns == NULL)
13833                 return (EINVAL);
13834         ipst = ns->netstack_ip;
13835         ASSERT(ipst != NULL);

13837         if (ipst->ips_ip_cgtp_filter) {
13838                 netstack_rele(ns);
13839                 return (EBUSY);
13840         }

13842         if (ipst->ips_ip_cgtp_filter_ops == NULL) {
13843                 netstack_rele(ns);
13844                 return (ENXIO);
13845         }
13846         ipst->ips_ip_cgtp_filter_ops = NULL;

13848         ill_set_inputfn_all(ipst);

13850         netstack_rele(ns);
13851         return (0);
13852 }

13854 /*
13855  * Check whether there is a CGTP filter registration.
```

```
 * Returns non-zero if there is a registration, otherwise returns zero.
 * Note: returns zero if bad stackid.
 */
int
ip_cgtp_filter_is_registered(netstackid_t stackid)
{
        netstack_t *ns;
        ip_stack_t *ipst;
        int ret;

        ns = netstack_find_by_stackid(stackid);
        if (ns == NULL)
                return (0);
        ipst = ns->netstack_ip;
        ASSERT(ipst != NULL);

        if (ipst->ips_ip_cgtp_filter_ops != NULL)
                ret = 1;
        else
                ret = 0;

        netstack_rele(ns);
        return (ret);
}

static int
ip_squeue_switch(int val)
{
        int rval;

        switch (val) {
        case IP_SQUEUE_ENTER_NODRAIN:
                rval = SQ_NODRAIN;
                break;
        case IP_SQUEUE_ENTER:
                rval = SQ_PROCESS;
                break;
        case IP_SQUEUE_FILL:
        default:
                rval = SQ_FILL;
                break;
        }
        return (rval);
}

static void *
ip_kstat2_init(netstackid_t stackid, ip_stat_t *ip_statisticsp)
{
        kstat_t *ksp;

        ip_stat_t template = {
                { "ip_udp_fannorm",             KSTAT_DATA_UINT64 },
                { "ip_udp_fanmb",               KSTAT_DATA_UINT64 },
                { "ip_recv_pullup",             KSTAT_DATA_UINT64 },
                { "ip_db_ref",                  KSTAT_DATA_UINT64 },
                { "ip_notaligned",              KSTAT_DATA_UINT64 },
                { "ip_multimblk",               KSTAT_DATA_UINT64 },
                { "ip_opt",                     KSTAT_DATA_UINT64 },
                { "ipsec_proto_ahesp",          KSTAT_DATA_UINT64 },
                { "ip_conn_flputbq",            KSTAT_DATA_UINT64 },
                { "ip_conn_walk_drain",         KSTAT_DATA_UINT64 },
                { "ip_out_sw_cksum",            KSTAT_DATA_UINT64 },
                { "ip_out_sw_cksum_bytes",      KSTAT_DATA_UINT64 },
                { "ip_in_sw_cksum",             KSTAT_DATA_UINT64 },
                { "ip_ire_reclaim_calls",       KSTAT_DATA_UINT64 },
                { "ip_ire_reclaim_deleted",     KSTAT_DATA_UINT64 },
```



```
                { "ip_nce_reclaim_calls",       KSTAT_DATA_UINT64 },
                { "ip_nce_reclaim_deleted",     KSTAT_DATA_UINT64 },
                { "ip_dce_reclaim_calls",       KSTAT_DATA_UINT64 },
                { "ip_dce_reclaim_deleted",     KSTAT_DATA_UINT64 },
                { "ip_tcp_in_full_hw_cksum_err",        KSTAT_DATA_UINT64 },
                { "ip_tcp_in_part_hw_cksum_err",        KSTAT_DATA_UINT64 },
                { "ip_tcp_in_sw_cksum_err",             KSTAT_DATA_UINT64 },
                { "ip_udp_in_full_hw_cksum_err",        KSTAT_DATA_UINT64 },
                { "ip_udp_in_part_hw_cksum_err",        KSTAT_DATA_UINT64 },
                { "ip_udp_in_sw_cksum_err",     KSTAT_DATA_UINT64 },
                { "conn_in_recvdstaddr",        KSTAT_DATA_UINT64 },
                { "conn_in_recvopts",           KSTAT_DATA_UINT64 },
                { "conn_in_recvif",             KSTAT_DATA_UINT64 },
                { "conn_in_recvslla",           KSTAT_DATA_UINT64 },
                { "conn_in_recvucred",          KSTAT_DATA_UINT64 },
                { "conn_in_recvttl",            KSTAT_DATA_UINT64 },
                { "conn_in_recvhopopts",        KSTAT_DATA_UINT64 },
                { "conn_in_recvhoplimit",       KSTAT_DATA_UINT64 },
                { "conn_in_recvdstopts",        KSTAT_DATA_UINT64 },
                { "conn_in_recvrthdrdstopts",   KSTAT_DATA_UINT64 },
                { "conn_in_recvrthdr",          KSTAT_DATA_UINT64 },
                { "conn_in_recvpktinfo",        KSTAT_DATA_UINT64 },
                { "conn_in_recvtclass",         KSTAT_DATA_UINT64 },
                { "conn_in_timestamp",          KSTAT_DATA_UINT64 },
        };

        ksp = kstat_create_netstack("ip", 0, "ipstat", "net",
            KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t),
            KSTAT_FLAG_VIRTUAL, stackid);

        if (ksp == NULL)
                return (NULL);

        bcopy(&template, ip_statisticsp, sizeof (template));
        ksp->ks_data = (void *)ip_statisticsp;
        ksp->ks_private = (void *)(uintptr_t)stackid;

        kstat_install(ksp);
        return (ksp);
}

static void
ip_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
{
        if (ksp != NULL) {
                ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
                kstat_delete_netstack(ksp, stackid);
        }
}

static void *
ip_kstat_init(netstackid_t stackid, ip_stack_t *ipst)
{
        kstat_t *ksp;

        ip_named_kstat_t template = {
                { "forwarding",         KSTAT_DATA_UINT32, 0 },
                { "defaultTTL",         KSTAT_DATA_UINT32, 0 },
                { "inReceives",         KSTAT_DATA_UINT64, 0 },
                { "inHdrErrors",        KSTAT_DATA_UINT32, 0 },
                { "inAddrErrors",       KSTAT_DATA_UINT32, 0 },
                { "forwDatagrams",      KSTAT_DATA_UINT64, 0 },
                { "inUnknownProtos",    KSTAT_DATA_UINT32, 0 },
                { "inDiscards",         KSTAT_DATA_UINT32, 0 },
                { "inDelivers",         KSTAT_DATA_UINT64, 0 },
                { "outRequests",        KSTAT_DATA_UINT64, 0 },
```

```
13988                 { "outDiscards",        KSTAT_DATA_UINT32, 0 },
13989                 { "outNoRoutes",        KSTAT_DATA_UINT32, 0 },
13990                 { "reasmTimeout",       KSTAT_DATA_UINT32, 0 },
13991                 { "reasmReqds",         KSTAT_DATA_UINT32, 0 },
13992                 { "reasmOKs",           KSTAT_DATA_UINT32, 0 },
13993                 { "reasmFails",         KSTAT_DATA_UINT32, 0 },
13994                 { "fragOKs",            KSTAT_DATA_UINT32, 0 },
13995                 { "fragFails",          KSTAT_DATA_UINT32, 0 },
13996                 { "fragCreates",        KSTAT_DATA_UINT32, 0 },
13997                 { "addrEntrySize",      KSTAT_DATA_INT32, 0 },
13998                 { "routeEntrySize",     KSTAT_DATA_INT32, 0 },
13999                 { "netToMediaEntrySize",        KSTAT_DATA_INT32, 0 },
14000                 { "routingDiscards",    KSTAT_DATA_UINT32, 0 },
14001                 { "inErrs",             KSTAT_DATA_UINT32, 0 },
14002                 { "noPorts",            KSTAT_DATA_UINT32, 0 },
14003                 { "inCksumErrs",        KSTAT_DATA_UINT32, 0 },
14004                 { "reasmDuplicates",    KSTAT_DATA_UINT32, 0 },
14005                 { "reasmPartDups",      KSTAT_DATA_UINT32, 0 },
14006                 { "forwProhibits",      KSTAT_DATA_UINT32, 0 },
14007                 { "udpInCksumErrs",     KSTAT_DATA_UINT32, 0 },
14008                 { "udpInOverflows",     KSTAT_DATA_UINT32, 0 },
14009                 { "rawipInOverflows",   KSTAT_DATA_UINT32, 0 },
14010                 { "ipsecInSucceeded",   KSTAT_DATA_UINT32, 0 },
14011                 { "ipsecInFailed",      KSTAT_DATA_INT32, 0 },
14012                 { "memberEntrySize",    KSTAT_DATA_INT32, 0 },
14013                 { "inIPv6",             KSTAT_DATA_UINT32, 0 },
14014                 { "outIPv6",            KSTAT_DATA_UINT32, 0 },
14015                 { "outSwitchIPv6",      KSTAT_DATA_UINT32, 0 },
14016         };

14018         ksp = kstat_create_netstack("ip", 0, "ip", "mib2", KSTAT_TYPE_NAMED,
14019             NUM_OF_FIELDS(ip_named_kstat_t), 0, stackid);
14020         if (ksp == NULL || ksp->ks_data == NULL)
14021                 return (NULL);

14023         template.forwarding.value.ui32 = WE_ARE_FORWARDING(ipst) ? 1:2;
14024         template.defaultTTL.value.ui32 = (uint32_t)ipst->ips_ip_def_ttl;
14025         template.reasmTimeout.value.ui32 = ipst->ips_ip_reassembly_timeout;
14026         template.addrEntrySize.value.i32 = sizeof (mib2_ipAddrEntry_t);
14027         template.routeEntrySize.value.i32 = sizeof (mib2_ipRouteEntry_t);

14029         template.netToMediaEntrySize.value.i32 =
14030             sizeof (mib2_ipNetToMediaEntry_t);

14032         template.memberEntrySize.value.i32 = sizeof (ipv6_member_t);

14034         bcopy(&template, ksp->ks_data, sizeof (template));
14035         ksp->ks_update = ip_kstat_update;
14036         ksp->ks_private = (void *)(uintptr_t)stackid;

14038         kstat_install(ksp);
14039         return (ksp);
14040 }

14042 static void
14043 ip_kstat_fini(netstackid_t stackid, kstat_t *ksp)
14044 {
14045         if (ksp != NULL) {
14046                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
14047                 kstat_delete_netstack(ksp, stackid);
14048         }
14049 }

14051 static int
14052 ip_kstat_update(kstat_t *kp, int rw)
14053 {
```



```
14054         ip_named_kstat_t *ipkp;
14055         mib2_ipIfStatsEntry_t ipmib;
14056         ill_walk_context_t ctx;
14057         ill_t *ill;
14058         netstackid_t    stackid = (zoneid_t)(uintptr_t)kp->ks_private;
14059         netstack_t      *ns;
14060         ip_stack_t      *ipst;

14062         if (kp == NULL || kp->ks_data == NULL)
14063                 return (EIO);

14065         if (rw == KSTAT_WRITE)
14066                 return (EACCES);

14068         ns = netstack_find_by_stackid(stackid);
14069         if (ns == NULL)
14070                 return (-1);
14071         ipst = ns->netstack_ip;
14072         if (ipst == NULL) {
14073                 netstack_rele(ns);
14074                 return (-1);
14075         }
14076         ipkp = (ip_named_kstat_t *)kp->ks_data;

14078         bcopy(&ipst->ips_ip_mib, &ipmib, sizeof (ipmib));
14079         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
14080         ill = ILL_START_WALK_V4(&ctx, ipst);
14081         for (; ill != NULL; ill = ill_next(&ctx, ill))
14082                 ip_mib2_add_ip_stats(&ipmib, ill->ill_ip_mib);
14083         rw_exit(&ipst->ips_ill_g_lock);

14085         ipkp->forwarding.value.ui32 =           ipmib.ipIfStatsForwarding;
14086         ipkp->defaultTTL.value.ui32 =           ipmib.ipIfStatsDefaultTTL;
14087         ipkp->inReceives.value.ui64 =           ipmib.ipIfStatsHCInReceives;
14088         ipkp->inHdrErrors.value.ui32 =          ipmib.ipIfStatsInHdrErrors;
14089         ipkp->inAddrErrors.value.ui32 =         ipmib.ipIfStatsInAddrErrors;
14090         ipkp->forwDatagrams.value.ui64 = ipmib.ipIfStatsHCOutForwDatagrams;
14091         ipkp->inUnknownProtos.value.ui32 =      ipmib.ipIfStatsInUnknownProtos;
14092         ipkp->inDiscards.value.ui32 =           ipmib.ipIfStatsInDiscards;
14093         ipkp->inDelivers.value.ui64 =           ipmib.ipIfStatsHCInDelivers;
14094         ipkp->outRequests.value.ui64 =          ipmib.ipIfStatsHCOutRequests;
14095         ipkp->outDiscards.value.ui32 =          ipmib.ipIfStatsOutDiscards;
14096         ipkp->outNoRoutes.value.ui32 =          ipmib.ipIfStatsOutNoRoutes;
14097         ipkp->reasmTimeout.value.ui32 =         ipst->ips_ip_reassembly_timeout;
14098         ipkp->reasmReqds.value.ui32 =           ipmib.ipIfStatsReasmReqds;
14099         ipkp->reasmOKs.value.ui32 =             ipmib.ipIfStatsReasmOKs;
14100         ipkp->reasmFails.value.ui32 =           ipmib.ipIfStatsReasmFails;
14101         ipkp->fragOKs.value.ui32 =              ipmib.ipIfStatsOutFragOKs;
14102         ipkp->fragFails.value.ui32 =            ipmib.ipIfStatsOutFragFails;
14103         ipkp->fragCreates.value.ui32 =          ipmib.ipIfStatsOutFragCreates;

14105         ipkp->routingDiscards.value.ui32 =      0;
14106         ipkp->inErrs.value.ui32 =               ipmib.tcpIfStatsInErrs;
14107         ipkp->noPorts.value.ui32 =              ipmib.udpIfStatsNoPorts;
14108         ipkp->inCksumErrs.value.ui32 =          ipmib.ipIfStatsInCksumErrs;
14109         ipkp->reasmDuplicates.value.ui32 =      ipmib.ipIfStatsReasmDuplicates;
14110         ipkp->reasmPartDups.value.ui32 =        ipmib.ipIfStatsReasmPartDups;
14111         ipkp->forwProhibits.value.ui32 =        ipmib.ipIfStatsForwProhibits;
14112         ipkp->udpInCksumErrs.value.ui32 =       ipmib.udpIfStatsInCksumErrs;
14113         ipkp->udpInOverflows.value.ui32 =       ipmib.udpIfStatsInOverflows;
14114         ipkp->rawipInOverflows.value.ui32 =     ipmib.rawipIfStatsInOverflows;
14115         ipkp->ipsecInSucceeded.value.ui32 =     ipmib.ipsecIfStatsInSucceeded;
14116         ipkp->ipsecInFailed.value.i32 =         ipmib.ipsecIfStatsInFailed;

14118         ipkp->inIPv6.value.ui32 =       ipmib.ipIfStatsInWrongIPVersion;
14119         ipkp->outIPv6.value.ui32 =      ipmib.ipIfStatsOutWrongIPVersion;
```

```
14120         ipkp->outSwitchIPv6.value.ui32 = ipmib.ipIfStatsOutSwitchIPVersion;

14122         netstack_rele(ns);

14124         return (0);
14125 }

14127 static void *
14128 icmp_kstat_init(netstackid_t stackid)
14129 {
14130         kstat_t *ksp;

14132         icmp_named_kstat_t template = {
14133                 { "inMsgs",             KSTAT_DATA_UINT32 },
14134                 { "inErrors",           KSTAT_DATA_UINT32 },
14135                 { "inDestUnreachs",     KSTAT_DATA_UINT32 },
14136                 { "inTimeExcds",        KSTAT_DATA_UINT32 },
14137                 { "inParmProbs",        KSTAT_DATA_UINT32 },
14138                 { "inSrcQuenchs",       KSTAT_DATA_UINT32 },
14139                 { "inRedirects",        KSTAT_DATA_UINT32 },
14140                 { "inEchos",            KSTAT_DATA_UINT32 },
14141                 { "inEchoReps",         KSTAT_DATA_UINT32 },
14142                 { "inTimestamps",       KSTAT_DATA_UINT32 },
14143                 { "inTimestampReps",    KSTAT_DATA_UINT32 },
14144                 { "inAddrMasks",        KSTAT_DATA_UINT32 },
14145                 { "inAddrMaskReps",     KSTAT_DATA_UINT32 },
14146                 { "outMsgs",            KSTAT_DATA_UINT32 },
14147                 { "outErrors",          KSTAT_DATA_UINT32 },
14148                 { "outDestUnreachs",    KSTAT_DATA_UINT32 },
14149                 { "outTimeExcds",       KSTAT_DATA_UINT32 },
14150                 { "outParmProbs",       KSTAT_DATA_UINT32 },
14151                 { "outSrcQuenchs",      KSTAT_DATA_UINT32 },
14152                 { "outRedirects",       KSTAT_DATA_UINT32 },
14153                 { "outEchos",           KSTAT_DATA_UINT32 },
14154                 { "outEchoReps",        KSTAT_DATA_UINT32 },
14155                 { "outTimestamps",      KSTAT_DATA_UINT32 },
14156                 { "outTimestampReps",   KSTAT_DATA_UINT32 },
14157                 { "outAddrMasks",       KSTAT_DATA_UINT32 },
14158                 { "outAddrMaskReps",    KSTAT_DATA_UINT32 },
14159                 { "inChksumErrs",       KSTAT_DATA_UINT32 },
14160                 { "inUnknowns",         KSTAT_DATA_UINT32 },
14161                 { "inFragNeeded",       KSTAT_DATA_UINT32 },
14162                 { "outFragNeeded",      KSTAT_DATA_UINT32 },
14163                 { "outDrops",           KSTAT_DATA_UINT32 },
14164                 { "inOverFlows",        KSTAT_DATA_UINT32 },
14165                 { "inBadRedirects",     KSTAT_DATA_UINT32 },
14166         };

14168         ksp = kstat_create_netstack("ip", 0, "icmp", "mib2", KSTAT_TYPE_NAMED,
14169             NUM_OF_FIELDS(icmp_named_kstat_t), 0, stackid);
14170         if (ksp == NULL || ksp->ks_data == NULL)
14171                 return (NULL);

14173         bcopy(&template, ksp->ks_data, sizeof (template));

14175         ksp->ks_update = icmp_kstat_update;
14176         ksp->ks_private = (void *)(uintptr_t)stackid;

14178         kstat_install(ksp);
14179         return (ksp);
14180 }

14182 static void
14183 icmp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
14184 {
14185         if (ksp != NULL) {
```



```
14186                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
14187                 kstat_delete_netstack(ksp, stackid);
14188         }
14189 }

14191 static int
14192 icmp_kstat_update(kstat_t *kp, int rw)
14193 {
14194         icmp_named_kstat_t *icmpkp;
14195         netstackid_t    stackid = (zoneid_t)(uintptr_t)kp->ks_private;
14196         netstack_t      *ns;
14197         ip_stack_t      *ipst;

14199         if ((kp == NULL) || (kp->ks_data == NULL))
14200                 return (EIO);

14202         if (rw == KSTAT_WRITE)
14203                 return (EACCES);

14205         ns = netstack_find_by_stackid(stackid);
14206         if (ns == NULL)
14207                 return (-1);
14208         ipst = ns->netstack_ip;
14209         if (ipst == NULL) {
14210                 netstack_rele(ns);
14211                 return (-1);
14212         }
14213         icmpkp = (icmp_named_kstat_t *)kp->ks_data;

14215         icmpkp->inMsgs.value.ui32 =         ipst->ips_icmp_mib.icmpInMsgs;
14216         icmpkp->inErrors.value.ui32 =       ipst->ips_icmp_mib.icmpInErrors;
14217         icmpkp->inDestUnreachs.value.ui32 =
14218             ipst->ips_icmp_mib.icmpInDestUnreachs;
14219         icmpkp->inTimeExcds.value.ui32 =    ipst->ips_icmp_mib.icmpInTimeExcds;
14220         icmpkp->inParmProbs.value.ui32 =    ipst->ips_icmp_mib.icmpInParmProbs;
14221         icmpkp->inSrcQuenchs.value.ui32 =   ipst->ips_icmp_mib.icmpInSrcQuenchs;
14222         icmpkp->inRedirects.value.ui32 =    ipst->ips_icmp_mib.icmpInRedirects;
14223         icmpkp->inEchos.value.ui32 =        ipst->ips_icmp_mib.icmpInEchos;
14224         icmpkp->inEchoReps.value.ui32 =     ipst->ips_icmp_mib.icmpInEchoReps;
14225         icmpkp->inTimestamps.value.ui32 =   ipst->ips_icmp_mib.icmpInTimestamps;
14226         icmpkp->inTimestampReps.value.ui32 =
14227             ipst->ips_icmp_mib.icmpInTimestampReps;
14228         icmpkp->inAddrMasks.value.ui32 =    ipst->ips_icmp_mib.icmpInAddrMasks;
14229         icmpkp->inAddrMaskReps.value.ui32 =
14230             ipst->ips_icmp_mib.icmpInAddrMaskReps;
14231         icmpkp->outMsgs.value.ui32 =        ipst->ips_icmp_mib.icmpOutMsgs;
14232         icmpkp->outErrors.value.ui32 =      ipst->ips_icmp_mib.icmpOutErrors;
14233         icmpkp->outDestUnreachs.value.ui32 =
14234             ipst->ips_icmp_mib.icmpOutDestUnreachs;
14235         icmpkp->outTimeExcds.value.ui32 =   ipst->ips_icmp_mib.icmpOutTimeExcds;
14236         icmpkp->outParmProbs.value.ui32 =   ipst->ips_icmp_mib.icmpOutParmProbs;
14237         icmpkp->outSrcQuenchs.value.ui32 =
14238             ipst->ips_icmp_mib.icmpOutSrcQuenchs;
14239         icmpkp->outRedirects.value.ui32 =   ipst->ips_icmp_mib.icmpOutRedirects;
14240         icmpkp->outEchos.value.ui32 =       ipst->ips_icmp_mib.icmpOutEchos;
14241         icmpkp->outEchoReps.value.ui32 =    ipst->ips_icmp_mib.icmpOutEchoReps;
14242         icmpkp->outTimestamps.value.ui32 =
14243             ipst->ips_icmp_mib.icmpOutTimestamps;
14244         icmpkp->outTimestampReps.value.ui32 =
14245             ipst->ips_icmp_mib.icmpOutTimestampReps;
14246         icmpkp->outAddrMasks.value.ui32 =
14247             ipst->ips_icmp_mib.icmpOutAddrMasks;
14248         icmpkp->outAddrMaskReps.value.ui32 =
14249             ipst->ips_icmp_mib.icmpOutAddrMaskReps;
14250         icmpkp->inCksumErrs.value.ui32 =    ipst->ips_icmp_mib.icmpInCksumErrs;
14251         icmpkp->inUnknowns.value.ui32 =     ipst->ips_icmp_mib.icmpInUnknowns;
```

```
14252         icmpkp->inFragNeeded.value.ui32 =   ipst->ips_icmp_mib.icmpInFragNeeded;
14253         icmpkp->outFragNeeded.value.ui32 =
14254             ipst->ips_icmp_mib.icmpOutFragNeeded;
14255         icmpkp->outDrops.value.ui32 =       ipst->ips_icmp_mib.icmpOutDrops;
14256         icmpkp->inOverflows.value.ui32 =    ipst->ips_icmp_mib.icmpInOverflows;
14257         icmpkp->inBadRedirects.value.ui32 =
14258             ipst->ips_icmp_mib.icmpInBadRedirects;

14260         netstack_rele(ns);
14261         return (0);
14262 }

14264 /*
14265  * This is the fanout function for raw socket opened for SCTP.  Note
14266  * that it is called after SCTP checks that there is no socket which
14267  * wants a packet.  Then before SCTP handles this out of the blue packet,
14268  * this function is called to see if there is any raw socket for SCTP.
14269  * If there is and it is bound to the correct address, the packet will
14270  * be sent to that socket.  Note that only one raw socket can be bound to
14271  * a port.  This is assured in ipcl_sctp_hash_insert();
14272  */
14273 void
14274 ip_fanout_sctp_raw(mblk_t *mp, ipha_t *ipha, ip6_t *ip6h, uint32_t ports,
14275     ip_recv_attr_t *ira)
14276 {
14277         conn_t          *connp;
14278         queue_t         *rq;
14279         boolean_t       secure;
14280         ill_t           *ill = ira->ira_ill;
14281         ip_stack_t      *ipst = ill->ill_ipst;
14282         ipsec_stack_t   *ipss = ipst->ips_netstack->netstack_ipsec;
14283         sctp_stack_t    *sctps = ipst->ips_netstack->netstack_sctp;
14284         iaflags_t       iraflags = ira->ira_flags;
14285         ill_t           *rill = ira->ira_rill;

14287         secure = iraflags & IRAF_IPSEC_SECURE;

14289         connp = ipcl_classify_raw(mp, IPPROTO_SCTP, ports, ipha, ip6h,
14290             ira, ipst);
14291         if (connp == NULL) {
14292                 /*
14293                  * Although raw sctp is not summed, OOB chunks must be.
14294                  * Drop the packet here if the sctp checksum failed.
14295                  */
14296                 if (iraflags & IRAF_SCTP_CSUM_ERR) {
14297                         SCTPS_BUMP_MIB(sctps, sctpChecksumError);
14298                         freemsg(mp);
14299                         return;
14300                 }
14301                 ira->ira_ill = ira->ira_rill = NULL;
14302                 sctp_ootb_input(mp, ira, ipst);
14303                 ira->ira_ill = ill;
14304                 ira->ira_rill = rill;
14305                 return;
14306         }
14307         rq = connp->conn_rq;
14308         if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrld : !canputnext(rq)) {
14309                 CONN_DEC_REF(connp);
14310                 BUMP_MIB(ill->ill_ip_mib, rawipIfStatsInOverflows);
14311                 freemsg(mp);
14312                 return;
14313         }
14314         if (((iraflags & IRAF_IS_IPV4) ?
14315             CONN_INBOUND_POLICY_PRESENT(connp, ipss) :
14316             CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)) ||
14317             secure) {
```



```
14318                 mp = ipsec_check_inbound_policy(mp, connp, ipha,
14319                     ip6h, ira);
14320                 if (mp == NULL) {
14321                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
14322                         /* Note that mp is NULL */
14323                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
14324                         CONN_DEC_REF(connp);
14325                         return;
14326                 }
14327         }

14329         if (iraflags & IRAF_ICMP_ERROR) {
14330                 (connp->conn_recvicmp)(connp, mp, NULL, ira);
14331         } else {
14332                 ill_t *rill = ira->ira_rill;

14334                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
14335                 /* This is the SOCK_RAW, IPPROTO_SCTP case. */
14336                 ira->ira_ill = ira->ira_rill = NULL;
14337                 (connp->conn_recv)(connp, mp, NULL, ira);
14338                 ira->ira_ill = ill;
14339                 ira->ira_rill = rill;
14340         }
14341         CONN_DEC_REF(connp);
14342 }

14344 /*
14345  * Free a packet that has the link-layer dl_unitdata_req_t or fast-path
14346  * header before the ip payload.
14347  */
14348 static void
14349 ip_xmit_flowctl_drop(ill_t *ill, mblk_t *mp, boolean_t is_fp_mp, int fp_mp_len)
14350 {
14351         int len = (mp->b_wptr - mp->b_rptr);
14352         mblk_t *ip_mp;

14354         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
14355         if (is_fp_mp || len != fp_mp_len) {
14356                 if (len > fp_mp_len) {
14357                         /*
14358                          * fastpath header and ip header in the first mblk
14359                          */
14360                         mp->b_rptr += fp_mp_len;
14361                 } else {
14362                         /*
14363                          * ip_xmit_attach_llhdr had to prepend an mblk to
14364                          * attach the fastpath header before ip header.
14365                          */
14366                         ip_mp = mp->b_cont;
14367                         freeb(mp);
14368                         mp = ip_mp;
14369                         mp->b_rptr += (fp_mp_len - len);
14370                 }
14371         } else {
14372                 ip_mp = mp->b_cont;
14373                 freeb(mp);
14374                 mp = ip_mp;
14375         }
14376         ip_drop_output("ipIfStatsOutDiscards - flow ctl", mp, ill);
14377         freemsg(mp);
14378 }

14380 /*
14381  * Normal post fragmentation function.
14382  *
14383  * Send a packet using the passed in nce. This handles both IPv4 and IPv6
```

```
 * using the same state machine.
 *
 * We return an error on failure. In particular we return EWOULDBLOCK
 * when the driver flow controls. In that case this ensures that ip_wsrv runs
 * (currently by canputnext failure resulting in backenabling from GLD.)
 * This allows the callers of conn_ip_output() to use EWOULDBLOCK as an
 * indication that they can flow control until ip_wsrv() tells then to restart.
 *
 * If the nce passed by caller is incomplete, this function
 * queues the packet and if necessary, sends ARP request and bails.
 * If the Neighbor Cache passed is fully resolved, we simply prepend
 * the link-layer header to the packet, do ipsec hw acceleration
 * work if necessary, and send the packet out on the wire.
 */
/* ARGSUSED6 */
int
ip_xmit(mblk_t *mp, nce_t *nce, iaflags_t ixaflags, uint_t pkt_len,
    uint32_t xmit_hint, zoneid_t szone, zoneid_t nolzid, uintptr_t *ixacookie)
{
        queue_t         *wq;
        ill_t           *ill = nce->nce_ill;
        ip_stack_t      *ipst = ill->ill_ipst;
        uint64_t        delta;
        boolean_t       isv6 = ill->ill_isv6;
        boolean_t       fp_mp;
        ncec_t          *ncec = nce->nce_common;
        int64_t         now = LBOLT_FASTPATH64;
        boolean_t       is_probe;

        DTRACE_PROBE1(ip__xmit, nce_t *, nce);

        ASSERT(mp != NULL);
        ASSERT(mp->b_datap->db_type == M_DATA);
        ASSERT(pkt_len == msgdsize(mp));

        /*
         * If we have already been here and are coming back after ARP/ND.
         * the IXAF_NO_TRACE flag is set. We skip FW_HOOKS, DTRACE and ipobs
         * in that case since they have seen the packet when it came here
         * the first time.
         */
        if (ixaflags & IXAF_NO_TRACE)
                goto sendit;

        if (ixaflags & IXAF_IS_IPV4) {
                ipha_t *ipha = (ipha_t *)mp->b_rptr;

                ASSERT(!isv6);
                ASSERT(pkt_len == ntohs(((ipha_t *)mp->b_rptr)->ipha_length));
                if (HOOKS4_INTERESTED_PHYSICAL_OUT(ipst) &&
                    !(ixaflags & IXAF_NO_PFHOOK)) {
                        int     error;

                        FW_HOOKS(ipst->ips_ip4_physical_out_event,
                            ipst->ips_ipv4firewall_physical_out,
                            NULL, ill, ipha, mp, mp, 0, ipst, error);
                        DTRACE_PROBE1(ip4__physical__out__end,
                            mblk_t *, mp);
                        if (mp == NULL)
                                return (error);

                        /* The length could have changed */
                        pkt_len = msgdsize(mp);
                }
                if (ipst->ips_ip4_observe.he_interested) {
                        /*
```



```
                         * Note that for TX the zoneid is the sending
                         * zone, whether or not MLP is in play.
                         * Since the szone argument is the IP zoneid (i.e.,
                         * zero for exclusive-IP zones) and ipobs wants
                         * the system zoneid, we map it here.
                         */
                        szone = IP_REAL_ZONEID(szone, ipst);

                        /*
                         * On the outbound path the destination zone will be
                         * unknown as we're sending this packet out on the
                         * wire.
                         */
                        ipobs_hook(mp, IPOBS_HOOK_OUTBOUND, szone, ALL_ZONES,
                            ill, ipst);
                }
                DTRACE_IP7(send, mblk_t *, mp,  conn_t *, NULL,
                    void_ip_t *, ipha,  __dtrace_ipsr_ill_t *, ill,
                    ipha_t *, ipha, ip6_t *, NULL, int, 0);
        } else {
                ip6_t *ip6h = (ip6_t *)mp->b_rptr;

                ASSERT(isv6);
                ASSERT(pkt_len ==
                    ntohs(((ip6_t *)mp->b_rptr)->ip6_plen) + IPV6_HDR_LEN);
                if (HOOKS6_INTERESTED_PHYSICAL_OUT(ipst) &&
                    !(ixaflags & IXAF_NO_PFHOOK)) {
                        int     error;

                        FW_HOOKS6(ipst->ips_ip6_physical_out_event,
                            ipst->ips_ipv6firewall_physical_out,
                            NULL, ill, ip6h, mp, mp, 0, ipst, error);
                        DTRACE_PROBE1(ip6__physical__out__end,
                            mblk_t *, mp);
                        if (mp == NULL)
                                return (error);

                        /* The length could have changed */
                        pkt_len = msgdsize(mp);
                }
                if (ipst->ips_ip6_observe.he_interested) {
                        /* See above */
                        szone = IP_REAL_ZONEID(szone, ipst);

                        ipobs_hook(mp, IPOBS_HOOK_OUTBOUND, szone, ALL_ZONES,
                            ill, ipst);
                }
                DTRACE_IP7(send, mblk_t *, mp,  conn_t *, NULL,
                    void_ip_t *, ip6h,  __dtrace_ipsr_ill_t *, ill,
                    ipha_t *, NULL, ip6_t *, ip6h, int, 0);
        }

sendit:
        /*
         * We check the state without a lock because the state can never
         * move "backwards" to initial or incomplete.
         */
        switch (ncec->ncec_state) {
        case ND_REACHABLE:
        case ND_STALE:
        case ND_DELAY:
        case ND_PROBE:
                mp = ip_xmit_attach_llhdr(mp, nce);
                if (mp == NULL) {
                        /*
                         * ip_xmit_attach_llhdr has increased
```

```
14516                          * ipIfStatsOutDiscards and called ip_drop_output()
14517                          */
14518                         return (ENOBUFS);
14519                 }
14520                 /*
14521                  * check if nce_fastpath completed and we tagged on a
14522                  * copy of nce_fp_mp in ip_xmit_attach_llhdr().
14523                  */
14524                 fp_mp = (mp->b_datap->db_type == M_DATA);

14526                 if (fp_mp &&
14527                     (ill->ill_capabilities & ILL_CAPAB_DLD_DIRECT)) {
14528                         ill_dld_direct_t *idd;

14530                         idd = &ill->ill_dld_capab->idc_direct;
14531                         /*
14532                          * Send the packet directly to DLD, where it
14533                          * may be queued depending on the availability
14534                          * of transmit resources at the media layer.
14535                          * Return value should be taken into
14536                          * account and flow control the TCP.
14537                          */
14538                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCOutTransmits);
14539                         UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCOutOctets,
14540                             pkt_len);

14542                         if (ixaflags & IXAF_NO_DEV_FLOW_CTL) {
14543                                 (void) idd->idd_tx_df(idd->idd_tx_dh, mp,
14544                                     (uintptr_t)xmit_hint, IP_DROP_ON_NO_DESC);
14545                         } else {
14546                                 uintptr_t cookie;

14548                                 if ((cookie = idd->idd_tx_df(idd->idd_tx_dh,
14549                                     mp, (uintptr_t)xmit_hint, 0)) != 0) {
14550                                         if (ixacookie != NULL)
14551                                                 *ixacookie = cookie;
14552                                         return (EWOULDBLOCK);
14553                                 }
14554                         }
14555                 } else {
14556                         wq = ill->ill_wq;

14558                         if (!(ixaflags & IXAF_NO_DEV_FLOW_CTL) &&
14559                             !canputnext(wq)) {
14560                                 if (ixacookie != NULL)
14561                                         *ixacookie = 0;
14562                                 ip_xmit_flowctl_drop(ill, mp, fp_mp,
14563                                     nce->nce_fp_mp != NULL ?
14564                                     MBLKL(nce->nce_fp_mp) : 0);
14565                                 return (EWOULDBLOCK);
14566                         }
14567                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCOutTransmits);
14568                         UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCOutOctets,
14569                             pkt_len);
14570                         putnext(wq, mp);
14571                 }

14573                 /*
14574                  * The rest of this function implements Neighbor Unreachability
14575                  * detection. Determine if the ncec is eligible for NUD.
14576                  */
14577                 if (ncec->ncec_flags & NCE_F_NONUD)
14578                         return (0);

14580                 ASSERT(ncec->ncec_state != ND_INCOMPLETE);
```



```
14582                 /*
14583                  * Check for upper layer advice
14584                  */
14585                 if (ixaflags & IXAF_REACH_CONF) {
14586                         timeout_id_t tid;

14588                         /*
14589                          * It should be o.k. to check the state without
14590                          * a lock here, at most we lose an advice.
14591                          */
14592                         ncec->ncec_last = TICK_TO_MSEC(now);
14593                         if (ncec->ncec_state != ND_REACHABLE) {
14594                                 mutex_enter(&ncec->ncec_lock);
14595                                 ncec->ncec_state = ND_REACHABLE;
14596                                 tid = ncec->ncec_timeout_id;
14597                                 ncec->ncec_timeout_id = 0;
14598                                 mutex_exit(&ncec->ncec_lock);
14599                                 (void) untimeout(tid);
14600                                 if (ip_debug > 2) {
14601                                         /* ip1dbg */
14602                                         pr_addr_dbg("ip_xmit: state"
14603                                             " for %s changed to"
14604                                             " REACHABLE\n", AF_INET6,
14605                                             &ncec->ncec_addr);
14606                                 }
14607                         }
14608                         return (0);
14609                 }

14611                 delta =  TICK_TO_MSEC(now) - ncec->ncec_last;
14612                 ip1dbg(("ip_xmit: delta = %" PRId64
14613                     " ill_reachable_time = %d \n", delta,
14614                     ill->ill_reachable_time));
14615                 if (delta > (uint64_t)ill->ill_reachable_time) {
14616                         mutex_enter(&ncec->ncec_lock);
14617                         switch (ncec->ncec_state) {
14618                         case ND_REACHABLE:
14619                                 ASSERT((ncec->ncec_flags & NCE_F_NONUD) == 0);
14620                                 /* FALLTHROUGH */
14621                         case ND_STALE:
14622                                 /*
14623                                  * ND_REACHABLE is identical to
14624                                  * ND_STALE in this specific case. If
14625                                  * reachable time has expired for this
14626                                  * neighbor (delta is greater than
14627                                  * reachable time), conceptually, the
14628                                  * neighbor cache is no longer in
14629                                  * REACHABLE state, but already in
14630                                  * STALE state.  So the correct
14631                                  * transition here is to ND_DELAY.
14632                                  */
14633                                 ncec->ncec_state = ND_DELAY;
14634                                 mutex_exit(&ncec->ncec_lock);
14635                                 nce_restart_timer(ncec,
14636                                     ipst->ips_delay_first_probe_time);
14637                                 if (ip_debug > 3) {
14638                                         /* ip2dbg */
14639                                         pr_addr_dbg("ip_xmit: state"
14640                                             " for %s changed to"
14641                                             " DELAY\n", AF_INET6,
14642                                             &ncec->ncec_addr);
14643                                 }
14644                                 break;
14645                         case ND_DELAY:
14646                         case ND_PROBE:
14647                                 mutex_exit(&ncec->ncec_lock);
```

```
14648                                   /* Timers have already started */
14649                                   break;
14650                           case ND_UNREACHABLE:
14651                                   /*
14652                                    * nce_timer has detected that this ncec
14653                                    * is unreachable and initiated deleting
14654                                    * this ncec.
14655                                    * This is a harmless race where we found the
14656                                    * ncec before it was deleted and have
14657                                    * just sent out a packet using this
14658                                    * unreachable ncec.
14659                                    */
14660                                   mutex_exit(&ncec->ncec_lock);
14661                                   break;
14662                           default:
14663                                   ASSERT(0);
14664                                   mutex_exit(&ncec->ncec_lock);
14665                           }
14666                   }
14667                   return (0);

14669           case ND_INCOMPLETE:
14670                   /*
14671                    * the state could have changed since we didn't hold the lock.
14672                    * Re-verify state under lock.
14673                    */
14674                   is_probe = ipmp_packet_is_probe(mp, nce->nce_ill);
14675                   mutex_enter(&ncec->ncec_lock);
14676                   if (NCE_ISREACHABLE(ncec)) {
14677                           mutex_exit(&ncec->ncec_lock);
14678                           goto sendit;
14679                   }
14680                   /* queue the packet */
14681                   nce_queue_mp(ncec, mp, is_probe);
14682                   mutex_exit(&ncec->ncec_lock);
14683                   DTRACE_PROBE2(ip__xmit__incomplete,
14684                       (ncec_t *), ncec, (mblk_t *), mp);
14685                   return (0);

14687           case ND_INITIAL:
14688                   /*
14689                    * State could have changed since we didn't hold the lock, so
14690                    * re-verify state.
14691                    */
14692                   is_probe = ipmp_packet_is_probe(mp, nce->nce_ill);
14693                   mutex_enter(&ncec->ncec_lock);
14694                   if (NCE_ISREACHABLE(ncec))  {
14695                           mutex_exit(&ncec->ncec_lock);
14696                           goto sendit;
14697                   }
14698                   nce_queue_mp(ncec, mp, is_probe);
14699                   if (ncec->ncec_state == ND_INITIAL) {
14700                           ncec->ncec_state = ND_INCOMPLETE;
14701                           mutex_exit(&ncec->ncec_lock);
14702                           /*
14703                            * figure out the source we want to use
14704                            * and resolve it.
14705                            */
14706                           ip_ndp_resolve(ncec);
14707                   } else  {
14708                           mutex_exit(&ncec->ncec_lock);
14709                   }
14710                   return (0);

14712           case ND_UNREACHABLE:
14713                   BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
```



```
14714                   ip_drop_output("ipIfStatsOutDiscards - ND_UNREACHABLE",
14715                       mp, ill);
14716                   freemsg(mp);
14717                   return (0);

14719           default:
14720                   ASSERT(0);
14721                   BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
14722                   ip_drop_output("ipIfStatsOutDiscards - ND_other",
14723                       mp, ill);
14724                   freemsg(mp);
14725                   return (ENETUNREACH);
14726           }
14727   }

14729   /*
14730    * Return B_TRUE if the buffers differ in length or content.
14731    * This is used for comparing extension header buffers.
14732    * Note that an extension header would be declared different
14733    * even if all that changed was the next header value in that header i.e.
14734    * what really changed is the next extension header.
14735    */
14736   boolean_t
14737   ip_cmpbuf(const void *abuf, uint_t alen, boolean_t b_valid, const void *bbuf,
14738       uint_t blen)
14739   {
14740           if (!b_valid)
14741                   blen = 0;

14743           if (alen != blen)
14744                   return (B_TRUE);
14745           if (alen == 0)
14746                   return (B_FALSE);       /* Both zero length */
14747           return (bcmp(abuf, bbuf, alen));
14748   }

14750   /*
14751    * Preallocate memory for ip_savebuf(). Returns B_TRUE if ok.
14752    * Return B_FALSE if memory allocation fails - don't change any state!
14753    */
14754   boolean_t
14755   ip_allocbuf(void **dstp, uint_t *dstlenp, boolean_t src_valid,
14756       const void *src, uint_t srclen)
14757   {
14758           void *dst;

14760           if (!src_valid)
14761                   srclen = 0;

14763           ASSERT(*dstlenp == 0);
14764           if (src != NULL && srclen != 0) {
14765                   dst = mi_alloc(srclen, BPRI_MED);
14766                   if (dst == NULL)
14767                           return (B_FALSE);
14768           } else {
14769                   dst = NULL;
14770           }
14771           if (*dstp != NULL)
14772                   mi_free(*dstp);
14773           *dstp = dst;
14774           *dstlenp = dst == NULL ? 0 : srclen;
14775           return (B_TRUE);
14776   }

14778   /*
14779    * Replace what is in *dst, *dstlen with the source.
```

```
 * Assumes ip_allocbuf has already been called.
 */
void
ip_savebuf(void **dstp, uint_t *dstlenp, boolean_t src_valid,
    const void *src, uint_t srclen)
{
        if (!src_valid)
                srclen = 0;

        ASSERT(*dstlenp == srclen);
        if (src != NULL && srclen != 0)
                bcopy(src, *dstp, srclen);
}

/*
 * Free the storage pointed to by the members of an ip_pkt_t.
 */
void
ip_pkt_free(ip_pkt_t *ipp)
{
        uint_t  fields = ipp->ipp_fields;

        if (fields & IPPF_HOPOPTS) {
                kmem_free(ipp->ipp_hopopts, ipp->ipp_hopoptslen);
                ipp->ipp_hopopts = NULL;
                ipp->ipp_hopoptslen = 0;
        }
        if (fields & IPPF_RTHDRDSTOPTS) {
                kmem_free(ipp->ipp_rthdrdstopts, ipp->ipp_rthdrdstoptslen);
                ipp->ipp_rthdrdstopts = NULL;
                ipp->ipp_rthdrdstoptslen = 0;
        }
        if (fields & IPPF_DSTOPTS) {
                kmem_free(ipp->ipp_dstopts, ipp->ipp_dstoptslen);
                ipp->ipp_dstopts = NULL;
                ipp->ipp_dstoptslen = 0;
        }
        if (fields & IPPF_RTHDR) {
                kmem_free(ipp->ipp_rthdr, ipp->ipp_rthdrlen);
                ipp->ipp_rthdr = NULL;
                ipp->ipp_rthdrlen = 0;
        }
        if (fields & IPPF_IPV4_OPTIONS) {
                kmem_free(ipp->ipp_ipv4_options, ipp->ipp_ipv4_options_len);
                ipp->ipp_ipv4_options = NULL;
                ipp->ipp_ipv4_options_len = 0;
        }
        if (fields & IPPF_LABEL_V4) {
                kmem_free(ipp->ipp_label_v4, ipp->ipp_label_len_v4);
                ipp->ipp_label_v4 = NULL;
                ipp->ipp_label_len_v4 = 0;
        }
        if (fields & IPPF_LABEL_V6) {
                kmem_free(ipp->ipp_label_v6, ipp->ipp_label_len_v6);
                ipp->ipp_label_v6 = NULL;
                ipp->ipp_label_len_v6 = 0;
        }
        ipp->ipp_fields &= ~(IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
            IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6);
}

/*
 * Copy from src to dst and allocate as needed.
 * Returns zero or ENOMEM.
 *
 * The caller must initialize dst to zero.
```



```
 */
int
ip_pkt_copy(ip_pkt_t *src, ip_pkt_t *dst, int kmflag)
{
        uint_t  fields = src->ipp_fields;

        /* Start with fields that don't require memory allocation */
        dst->ipp_fields = fields &
            ~(IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
            IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6);

        dst->ipp_addr = src->ipp_addr;
        dst->ipp_unicast_hops = src->ipp_unicast_hops;
        dst->ipp_hoplimit = src->ipp_hoplimit;
        dst->ipp_tclass = src->ipp_tclass;
        dst->ipp_type_of_service = src->ipp_type_of_service;

        if (!(fields & (IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
            IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6)))
                return (0);

        if (fields & IPPF_HOPOPTS) {
                dst->ipp_hopopts = kmem_alloc(src->ipp_hopoptslen, kmflag);
                if (dst->ipp_hopopts == NULL) {
                        ip_pkt_free(dst);
                        return (ENOMEM);
                }
                dst->ipp_fields |= IPPF_HOPOPTS;
                bcopy(src->ipp_hopopts, dst->ipp_hopopts,
                    src->ipp_hopoptslen);
                dst->ipp_hopoptslen = src->ipp_hopoptslen;
        }
        if (fields & IPPF_RTHDRDSTOPTS) {
                dst->ipp_rthdrdstopts = kmem_alloc(src->ipp_rthdrdstoptslen,
                    kmflag);
                if (dst->ipp_rthdrdstopts == NULL) {
                        ip_pkt_free(dst);
                        return (ENOMEM);
                }
                dst->ipp_fields |= IPPF_RTHDRDSTOPTS;
                bcopy(src->ipp_rthdrdstopts, dst->ipp_rthdrdstopts,
                    src->ipp_rthdrdstoptslen);
                dst->ipp_rthdrdstoptslen = src->ipp_rthdrdstoptslen;
        }
        if (fields & IPPF_DSTOPTS) {
                dst->ipp_dstopts = kmem_alloc(src->ipp_dstoptslen, kmflag);
                if (dst->ipp_dstopts == NULL) {
                        ip_pkt_free(dst);
                        return (ENOMEM);
                }
                dst->ipp_fields |= IPPF_DSTOPTS;
                bcopy(src->ipp_dstopts, dst->ipp_dstopts,
                    src->ipp_dstoptslen);
                dst->ipp_dstoptslen = src->ipp_dstoptslen;
        }
        if (fields & IPPF_RTHDR) {
                dst->ipp_rthdr = kmem_alloc(src->ipp_rthdrlen, kmflag);
                if (dst->ipp_rthdr == NULL) {
                        ip_pkt_free(dst);
                        return (ENOMEM);
                }
                dst->ipp_fields |= IPPF_RTHDR;
                bcopy(src->ipp_rthdr, dst->ipp_rthdr,
                    src->ipp_rthdrlen);
                dst->ipp_rthdrlen = src->ipp_rthdrlen;
        }
```

```
14912         if (fields & IPPF_IPV4_OPTIONS) {
14913                 dst->ipp_ipv4_options = kmem_alloc(src->ipp_ipv4_options_len,
14914                     kmflag);
14915                 if (dst->ipp_ipv4_options == NULL) {
14916                         ip_pkt_free(dst);
14917                         return (ENOMEM);
14918                 }
14919                 dst->ipp_fields |= IPPF_IPV4_OPTIONS;
14920                 bcopy(src->ipp_ipv4_options, dst->ipp_ipv4_options,
14921                     src->ipp_ipv4_options_len);
14922                 dst->ipp_ipv4_options_len = src->ipp_ipv4_options_len;
14923         }
14924         if (fields & IPPF_LABEL_V4) {
14925                 dst->ipp_label_v4 = kmem_alloc(src->ipp_label_len_v4, kmflag);
14926                 if (dst->ipp_label_v4 == NULL) {
14927                         ip_pkt_free(dst);
14928                         return (ENOMEM);
14929                 }
14930                 dst->ipp_fields |= IPPF_LABEL_V4;
14931                 bcopy(src->ipp_label_v4, dst->ipp_label_v4,
14932                     src->ipp_label_len_v4);
14933                 dst->ipp_label_len_v4 = src->ipp_label_len_v4;
14934         }
14935         if (fields & IPPF_LABEL_V6) {
14936                 dst->ipp_label_v6 = kmem_alloc(src->ipp_label_len_v6, kmflag);
14937                 if (dst->ipp_label_v6 == NULL) {
14938                         ip_pkt_free(dst);
14939                         return (ENOMEM);
14940                 }
14941                 dst->ipp_fields |= IPPF_LABEL_V6;
14942                 bcopy(src->ipp_label_v6, dst->ipp_label_v6,
14943                     src->ipp_label_len_v6);
14944                 dst->ipp_label_len_v6 = src->ipp_label_len_v6;
14945         }
14946         if (fields & IPPF_FRAGHDR) {
14947                 dst->ipp_fraghdr = kmem_alloc(src->ipp_fraghdrlen, kmflag);
14948                 if (dst->ipp_fraghdr == NULL) {
14949                         ip_pkt_free(dst);
14950                         return (ENOMEM);
14951                 }
14952                 dst->ipp_fields |= IPPF_FRAGHDR;
14953                 bcopy(src->ipp_fraghdr, dst->ipp_fraghdr,
14954                     src->ipp_fraghdrlen);
14955                 dst->ipp_fraghdrlen = src->ipp_fraghdrlen;
14956         }
14957         return (0);
14958 }

14960 /*
14961  * Returns INADDR_ANY if no source route
14962  */
14963 ipaddr_t
14964 ip_pkt_source_route_v4(const ip_pkt_t *ipp)
14965 {
14966         ipaddr_t        nexthop = INADDR_ANY;
14967         ipoptp_t        opts;
14968         uchar_t         *opt;
14969         uint8_t         optval;
14970         uint8_t         optlen;
14971         uint32_t        totallen;

14973         if (!(ipp->ipp_fields & IPPF_IPV4_OPTIONS))
14974                 return (INADDR_ANY);

14976         totallen = ipp->ipp_ipv4_options_len;
14977         if (totallen & 0x3)
```



```
14978                 return (INADDR_ANY);

14980         for (optval = ipoptp_first2(&opts, totallen, ipp->ipp_ipv4_options);
14981             optval != IPOPT_EOL;
14982             optval = ipoptp_next(&opts)) {
14983                 opt = opts.ipoptp_cur;
14984                 switch (optval) {
14985                         uint8_t off;
14986                 case IPOPT_SSRR:
14987                 case IPOPT_LSRR:
14988                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
14989                                 break;
14990                         }
14991                         optlen = opts.ipoptp_len;
14992                         off = opt[IPOPT_OFFSET];
14993                         off--;
14994                         if (optlen < IP_ADDR_LEN ||
14995                             off > optlen - IP_ADDR_LEN) {
14996                                 /* End of source route */
14997                                 break;
14998                         }
14999                         bcopy((char *)opt + off, &nexthop, IP_ADDR_LEN);
15000                         if (nexthop == htonl(INADDR_LOOPBACK)) {
15001                                 /* Ignore */
15002                                 nexthop = INADDR_ANY;
15003                                 break;
15004                         }
15005                         break;
15006                 }
15007         }
15008         return (nexthop);
15009 }

15011 /*
15012  * Reverse a source route.
15013  */
15014 void
15015 ip_pkt_source_route_reverse_v4(ip_pkt_t *ipp)
15016 {
15017         ipaddr_t        tmp;
15018         ipoptp_t        opts;
15019         uchar_t         *opt;
15020         uint8_t         optval;
15021         uint32_t        totallen;

15023         if (!(ipp->ipp_fields & IPPF_IPV4_OPTIONS))
15024                 return;

15026         totallen = ipp->ipp_ipv4_options_len;
15027         if (totallen & 0x3)
15028                 return;

15030         for (optval = ipoptp_first2(&opts, totallen, ipp->ipp_ipv4_options);
15031             optval != IPOPT_EOL;
15032             optval = ipoptp_next(&opts)) {
15033                 uint8_t off1, off2;

15035                 opt = opts.ipoptp_cur;
15036                 switch (optval) {
15037                 case IPOPT_SSRR:
15038                 case IPOPT_LSRR:
15039                         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
15040                                 break;
15041                         }
15042                         off1 = IPOPT_MINOFF_SR - 1;
15043                         off2 = opt[IPOPT_OFFSET] - IP_ADDR_LEN - 1;
```

```
15044                         while (off2 > off1) {
15045                                 bcopy(opt + off2, &tmp, IP_ADDR_LEN);
15046                                 bcopy(opt + off1, opt + off2, IP_ADDR_LEN);
15047                                 bcopy(&tmp, opt + off2, IP_ADDR_LEN);
15048                                 off2 -= IP_ADDR_LEN;
15049                                 off1 += IP_ADDR_LEN;
15050                         }
15051                         opt[IPOPT_OFFSET] = IPOPT_MINOFF_SR;
15052                         break;
15053                 }
15054         }
15055 }

15057 /*
15058  * Returns NULL if no routing header
15059  */
15060 in6_addr_t *
15061 ip_pkt_source_route_v6(const ip_pkt_t *ipp)
15062 {
15063         in6_addr_t      *nexthop = NULL;
15064         ip6_rthdr0_t    *rthdr;

15066         if (!(ipp->ipp_fields & IPPF_RTHDR))
15067                 return (NULL);

15069         rthdr = (ip6_rthdr0_t *)ipp->ipp_rthdr;
15070         if (rthdr->ip6r0_segleft == 0)
15071                 return (NULL);

15073         nexthop = (in6_addr_t *)((char *)rthdr + sizeof (*rthdr));
15074         return (nexthop);
15075 }

15077 zoneid_t
15078 ip_get_zoneid_v4(ipaddr_t addr, mblk_t *mp, ip_recv_attr_t *ira,
15079     zoneid_t lookup_zoneid)
15080 {
15081         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
15082         ire_t           *ire;
15083         int             ire_flags = MATCH_IRE_TYPE;
15084         zoneid_t        zoneid = ALL_ZONES;

15086         if (is_system_labeled() && !tsol_can_accept_raw(mp, ira, B_FALSE))
15087                 return (ALL_ZONES);

15089         if (lookup_zoneid != ALL_ZONES)
15090                 ire_flags |= MATCH_IRE_ZONEONLY;
15091         ire = ire_ftable_lookup_v4(addr, NULL, NULL, IRE_LOCAL | IRE_LOOPBACK,
15092             NULL, lookup_zoneid, NULL, ire_flags, 0, ipst, NULL);
15093         if (ire != NULL) {
15094                 zoneid = IP_REAL_ZONEID(ire->ire_zoneid, ipst);
15095                 ire_refrele(ire);
15096         }
15097         return (zoneid);
15098 }

15100 zoneid_t
15101 ip_get_zoneid_v6(in6_addr_t *addr, mblk_t *mp, const ill_t *ill,
15102     ip_recv_attr_t *ira, zoneid_t lookup_zoneid)
15103 {
15104         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
15105         ire_t           *ire;
15106         int             ire_flags = MATCH_IRE_TYPE;
15107         zoneid_t        zoneid = ALL_ZONES;

15109         if (is_system_labeled() && !tsol_can_accept_raw(mp, ira, B_FALSE))
```



```
15110                 return (ALL_ZONES);

15112         if (IN6_IS_ADDR_LINKLOCAL(addr))
15113                 ire_flags |= MATCH_IRE_ILL;

15115         if (lookup_zoneid != ALL_ZONES)
15116                 ire_flags |= MATCH_IRE_ZONEONLY;
15117         ire = ire_ftable_lookup_v6(addr, NULL, NULL, IRE_LOCAL | IRE_LOOPBACK,
15118             ill, lookup_zoneid, NULL, ire_flags, 0, ipst, NULL);
15119         if (ire != NULL) {
15120                 zoneid = IP_REAL_ZONEID(ire->ire_zoneid, ipst);
15121                 ire_refrele(ire);
15122         }
15123         return (zoneid);
15124 }

15126 /*
15127  * IP obserability hook support functions.
15128  */
15129 static void
15130 ipobs_init(ip_stack_t *ipst)
15131 {
15132         netid_t id;

15134         id = net_getnetidbynetstackid(ipst->ips_netstack->netstack_stackid);

15136         ipst->ips_ip4_observe_pr = net_protocol_lookup(id, NHF_INET);
15137         VERIFY(ipst->ips_ip4_observe_pr != NULL);

15139         ipst->ips_ip6_observe_pr = net_protocol_lookup(id, NHF_INET6);
15140         VERIFY(ipst->ips_ip6_observe_pr != NULL);
15141 }

15143 static void
15144 ipobs_fini(ip_stack_t *ipst)
15145 {

15147         VERIFY(net_protocol_release(ipst->ips_ip4_observe_pr) == 0);
15148         VERIFY(net_protocol_release(ipst->ips_ip6_observe_pr) == 0);
15149 }

15151 /*
15152  * hook_pkt_observe_t is composed in network byte order so that the
15153  * entire mblk_t chain handed into hook_run can be used as-is.
15154  * The caveat is that use of the fields, such as the zone fields,
15155  * requires conversion into host byte order first.
15156  */
15157 void
15158 ipobs_hook(mblk_t *mp, int htype, zoneid_t zsrc, zoneid_t zdst,
15159     const ill_t *ill, ip_stack_t *ipst)
15160 {
15161         hook_pkt_observe_t *hdr;
15162         uint64_t grifindex;
15163         mblk_t *imp;

15165         imp = allocb(sizeof (*hdr), BPRI_HI);
15166         if (imp == NULL)
15167                 return;

15169         hdr = (hook_pkt_observe_t *)imp->b_rptr;
15170         /*
15171          * b_wptr is set to make the apparent size of the data in the mblk_t
15172          * to exclude the pointers at the end of hook_pkt_observer_t.
15173          */
15174         imp->b_wptr = imp->b_rptr + sizeof (dl_ipnetinfo_t);
15175         imp->b_cont = mp;
```

```
15177         ASSERT(DB_TYPE(mp) == M_DATA);

15179         if (IS_UNDER_IPMP(ill))
15180                 grifindex = ipmp_ill_get_ipmp_ifindex(ill);
15181         else
15182                 grifindex = 0;

15184         hdr->hpo_version = 1;
15185         hdr->hpo_htype = htons(htype);
15186         hdr->hpo_pktlen = htonl((ulong_t)msgdsize(mp));
15187         hdr->hpo_ifindex = htonl(ill->ill_phyint->phyint_ifindex);
15188         hdr->hpo_grifindex = htonl(grifindex);
15189         hdr->hpo_zsrc = htonl(zsrc);
15190         hdr->hpo_zdst = htonl(zdst);
15191         hdr->hpo_pkt = imp;
15192         hdr->hpo_ctx = ipst->ips_netstack;

15194         if (ill->ill_isv6) {
15195                 hdr->hpo_family = AF_INET6;
15196                 (void) hook_run(ipst->ips_ipv6_net_data->netd_hooks,
15197                     ipst->ips_ipv6observing, (hook_data_t)hdr);
15198         } else {
15199                 hdr->hpo_family = AF_INET;
15200                 (void) hook_run(ipst->ips_ipv4_net_data->netd_hooks,
15201                     ipst->ips_ipv4observing, (hook_data_t)hdr);
15202         }

15204         imp->b_cont = NULL;
15205         freemsg(imp);
15206 }

15208 /*
15209  * Utility routine that checks if 'v4srcp' is a valid address on underlying
15210  * interface 'ill'.  If 'ipifp' is non-NULL, it's set to a held ipif
15211  * associated with 'v4srcp' on success.  NOTE: if this is not called from
15212  * inside the IPSQ (ill_g_lock is not held), 'ill' may be removed from the
15213  * group during or after this lookup.
15214  */
15215 boolean_t
15216 ipif_lookup_testaddr_v4(ill_t *ill, const in_addr_t *v4srcp, ipif_t **ipifp)
15217 {
15218         ipif_t *ipif;

15220         ipif = ipif_lookup_addr_exact(*v4srcp, ill, ill->ill_ipst);
15221         if (ipif != NULL) {
15222                 if (ipifp != NULL)
15223                         *ipifp = ipif;
15224                 else
15225                         ipif_refrele(ipif);
15226                 return (B_TRUE);
15227         }

15229         ip1dbg(("ipif_lookup_testaddr_v4: cannot find ipif for src %x\n",
15230             *v4srcp));
15231         return (B_FALSE);
15232 }

15234 /*
15235  * Transport protocol call back function for CPU state change.
15236  */
15237 /* ARGSUSED */
15238 static int
15239 ip_tp_cpu_update(cpu_setup_t what, int id, void *arg)
15240 {
15241         processorid_t cpu_seqid;
```



```
15242         netstack_handle_t nh;
15243         netstack_t *ns;

15245         ASSERT(MUTEX_HELD(&cpu_lock));

15247         switch (what) {
15248         case CPU_CONFIG:
15249         case CPU_ON:
15250         case CPU_INIT:
15251         case CPU_CPUPART_IN:
15252                 cpu_seqid = cpu[id]->cpu_seqid;
15253                 netstack_next_init(&nh);
15254                 while ((ns = netstack_next(&nh)) != NULL) {
15255                         dccp_stack_cpu_add(ns->netstack_dccp, cpu_seqid);
15256 #endif /* ! codereview */
15257                         tcp_stack_cpu_add(ns->netstack_tcp, cpu_seqid);
15258                         sctp_stack_cpu_add(ns->netstack_sctp, cpu_seqid);
15259                         udp_stack_cpu_add(ns->netstack_udp, cpu_seqid);
15260                         netstack_rele(ns);
15261                 }
15262                 netstack_next_fini(&nh);
15263                 break;
15264         case CPU_UNCONFIG:
15265         case CPU_OFF:
15266         case CPU_CPUPART_OUT:
15267                 /*
15268                  * Nothing to do.  We don't remove the per CPU stats from
15269                  * the IP stack even when the CPU goes offline.
15270                  */
15271                 break;
15272         default:
15273                 break;
15274         }
15275         return (0);
15276 }
```

```
**********************************************************
  533855 Mon Jul  9 14:38:16 2012
new/usr/src/uts/common/inet/ip/ip_if.c
dccp: properties
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 1990 Mentat Inc.
 */

/*
 * This file contains the interface control functions for IP.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/dlpi.h>
#include <sys/stropts.h>
#include <sys/strsun.h>
#include <sys/sysmacros.h>
#include <sys/strsubr.h>
#include <sys/strlog.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cmn_err.h>
#include <sys/kstat.h>
#include <sys/debug.h>
#include <sys/zone.h>
#include <sys/sunldi.h>
#include <sys/file.h>
#include <sys/bitmap.h>
#include <sys/cpuvar.h>
#include <sys/time.h>
#include <sys/ctype.h>
#include <sys/kmem.h>
#include <sys/systm.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/isa_defs.h>
#include <net/if.h>
#include <net/if_arp.h>
#include <net/if_types.h>
#include <net/if_dl.h>
#include <net/route.h>
#include <sys/sockio.h>
#include <netinet/in.h>
```



```
#include <netinet/ip6.h>
#include <netinet/icmp6.h>
#include <netinet/igmp_var.h>
#include <sys/policy.h>
#include <sys/ethernet.h>
#include <sys/callb.h>
#include <sys/md5.h>

#include <inet/common.h>   /* for various inet/mi.h and inet/nd.h needs */
#include <inet/mi.h>
#include <inet/nd.h>
#include <inet/tunables.h>
#include <inet/arp.h>
#include <inet/ip_arp.h>
#include <inet/mib2.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/ip6_asp.h>
#include <inet/tcp.h>
#include <inet/ip_multi.h>
#include <inet/ip_ire.h>
#include <inet/ip_ftable.h>
#include <inet/ip_rts.h>
#include <inet/ip_ndp.h>
#include <inet/ip_if.h>
#include <inet/ip_impl.h>
#include <inet/sctp_ip.h>
#include <inet/ip_netinfo.h>
#include <inet/ilb_ip.h>

#include <netinet/igmp.h>
#include <inet/ip_listutils.h>
#include <inet/ipclassifier.h>
#include <sys/mac_client.h>
#include <sys/dld.h>
#include <sys/mac_flow.h>

#include <sys/systeminfo.h>
#include <sys/bootconf.h>

#include <sys/tsol/tndb.h>
#include <sys/tsol/tnet.h>

#include <inet/rawip_impl.h> /* needed for icmp_stack_t */
#include <inet/udp_impl.h> /* needed for udp_stack_t */
#include <inet/dccp/dccp_stack.h> /* needed for dccp_stack_t */
#endif /* ! codereview */

/* The character which tells where the ill_name ends */
#define IPIF_SEPARATOR_CHAR     ':'

/* IP ioctl function table entry */
typedef struct ipft_s {
        int     ipft_cmd;
        pfi_t   ipft_pfi;
        int     ipft_min_size;
        int     ipft_flags;
} ipft_t;
#define IPFT_F_NO_REPLY         0x1     /* IP ioctl does not expect any reply */
#define IPFT_F_SELF_REPLY       0x2     /* ioctl callee does the ioctl reply */

static int      nd_ill_forward_get(queue_t *, mblk_t *, caddr_t, cred_t *);
static int      nd_ill_forward_set(queue_t *q, mblk_t *mp,
                    char *value, caddr_t cp, cred_t *ioc_cr);

static boolean_t ill_is_quiescent(ill_t *);
```

```
128 static boolean_t ip_addr_ok_v4(ipaddr_t addr, ipaddr_t subnet_mask);
129 static ip_m_t   *ip_m_lookup(t_uscalar_t mac_type);
130 static int      ip_sioctl_addr_tail(ipif_t *ipif, sin_t *sin, queue_t *q,
131     mblk_t *mp, boolean_t need_up);
132 static int      ip_sioctl_dstaddr_tail(ipif_t *ipif, sin_t *sin, queue_t *q,
133     mblk_t *mp, boolean_t need_up);
134 static int      ip_sioctl_slifzone_tail(ipif_t *ipif, zoneid_t zoneid,
135     queue_t *q, mblk_t *mp, boolean_t need_up);
136 static int      ip_sioctl_flags_tail(ipif_t *ipif, uint64_t flags, queue_t *q,
137     mblk_t *mp);
138 static int      ip_sioctl_netmask_tail(ipif_t *ipif, sin_t *sin, queue_t *q,
139     mblk_t *mp);
140 static int      ip_sioctl_subnet_tail(ipif_t *ipif, in6_addr_t, in6_addr_t,
141     queue_t *q, mblk_t *mp, boolean_t need_up);
142 static int      ip_sioctl_plink_ipmod(ipsq_t *ipsq, queue_t *q, mblk_t *mp,
143     int ioccmd, struct linkblk *li);
144 static ipaddr_t ip_subnet_mask(ipaddr_t addr, ipif_t **, ip_stack_t *);
145 static void     ip_wput_ioctl(queue_t *q, mblk_t *mp);
146 static void     ipsq_flush(ill_t *ill);

148 static  int     ip_sioctl_token_tail(ipif_t *ipif, sin6_t *sin6, int addrlen,
149     queue_t *q, mblk_t *mp, boolean_t need_up);
150 static void     ipsq_delete(ipsq_t *);

152 static ipif_t   *ipif_allocate(ill_t *ill, int id, uint_t ire_type,
153     boolean_t initialize, boolean_t insert, int *errorp);
154 static ire_t    **ipif_create_bcast_ires(ipif_t *ipif, ire_t **irep);
155 static void     ipif_delete_bcast_ires(ipif_t *ipif);
156 static int      ipif_add_ires_v4(ipif_t *, boolean_t);
157 static boolean_t ipif_comp_multi(ipif_t *old_ipif, ipif_t *new_ipif,
158                     boolean_t isv6);
159 static int      ipif_logical_down(ipif_t *ipif, queue_t *q, mblk_t *mp);
160 static void     ipif_free(ipif_t *ipif);
161 static void     ipif_free_tail(ipif_t *ipif);
162 static void     ipif_set_default(ipif_t *ipif);
163 static int      ipif_set_values(queue_t *q, mblk_t *mp,
164     char *interf_name, uint_t *ppa);
165 static int      ipif_set_values_tail(ill_t *ill, ipif_t *ipif, mblk_t *mp,
166     queue_t *q);
167 static ipif_t   *ipif_lookup_on_name(char *name, size_t namelen,
168     boolean_t do_alloc, boolean_t *exists, boolean_t isv6, zoneid_t zoneid,
169     ip_stack_t *);
170 static ipif_t   *ipif_lookup_on_name_async(char *name, size_t namelen,
171     boolean_t isv6, zoneid_t zoneid, queue_t *q, mblk_t *mp, ipsq_func_t func,
172     int *error, ip_stack_t *);

174 static int      ill_alloc_ppa(ill_if_t *, ill_t *);
175 static void     ill_delete_interface_type(ill_if_t *);
176 static int      ill_dl_up(ill_t *ill, ipif_t *ipif, mblk_t *mp, queue_t *q);
177 static void     ill_dl_down(ill_t *ill);
178 static void     ill_down(ill_t *ill);
179 static void     ill_down_ipifs(ill_t *, boolean_t);
180 static void     ill_free_mib(ill_t *ill);
181 static void     ill_glist_delete(ill_t *);
182 static void     ill_phyint_reinit(ill_t *ill);
183 static void     ill_set_nce_router_flags(ill_t *, boolean_t);
184 static void     ill_set_phys_addr_tail(ipsq_t *, queue_t *, mblk_t *, void *);
185 static void     ill_replumb_tail(ipsq_t *, queue_t *, mblk_t *, void *);

187 static ip_v6intfid_func_t ip_ether_v6intfid, ip_ib_v6intfid;
188 static ip_v6intfid_func_t ip_ipv4_v6intfid, ip_ipv6_v6intfid;
189 static ip_v6intfid_func_t ip_ipmp_v6intfid, ip_nodef_v6intfid;
190 static ip_v6intfid_func_t ip_ipv4_v6destintfid, ip_ipv6_v6destintfid;
191 static ip_v4mapinfo_func_t ip_ether_v4_mapping;
192 static ip_v6mapinfo_func_t ip_ether_v6_mapping;
193 static ip_v4mapinfo_func_t ip_ib_v4_mapping;
```



```
194 static ip_v6mapinfo_func_t ip_ib_v6_mapping;
195 static ip_v4mapinfo_func_t ip_mbcast_mapping;
196 static void     ip_cgtp_bcast_add(ire_t *, ip_stack_t *);
197 static void     ip_cgtp_bcast_delete(ire_t *, ip_stack_t *);
198 static void     phyint_free(phyint_t *);

200 static void ill_capability_dispatch(ill_t *, mblk_t *, dl_capability_sub_t *);
201 static void ill_capability_id_ack(ill_t *, mblk_t *, dl_capability_sub_t *);
202 static void ill_capability_vrrp_ack(ill_t *, mblk_t *, dl_capability_sub_t *);
203 static void ill_capability_hcksum_ack(ill_t *, mblk_t *, dl_capability_sub_t *);
204 static void ill_capability_hcksum_reset_fill(ill_t *, mblk_t *);
205 static void ill_capability_zerocopy_ack(ill_t *, mblk_t *,
206     dl_capability_sub_t *);
207 static void ill_capability_zerocopy_reset_fill(ill_t *, mblk_t *);
208 static void     ill_capability_dld_reset_fill(ill_t *, mblk_t *);
209 static void     ill_capability_dld_ack(ill_t *, mblk_t *,
210                     dl_capability_sub_t *);
211 static void     ill_capability_dld_enable(ill_t *);
212 static void     ill_capability_ack_thr(void *);
213 static void     ill_capability_lso_enable(ill_t *);

215 static ill_t    *ill_prev_usesrc(ill_t *);
216 static int      ill_relink_usesrc_ills(ill_t *, ill_t *, uint_t);
217 static void     ill_disband_usesrc_group(ill_t *);
218 static void     ip_sioctl_garp_reply(mblk_t *, ill_t *, void *, int);

220 #ifdef DEBUG
221 static  void    ill_trace_cleanup(const ill_t *);
222 static  void    ipif_trace_cleanup(const ipif_t *);
223 #endif

225 static  void    ill_dlpi_clear_deferred(ill_t *ill);

227 /*
228  * if we go over the memory footprint limit more than once in this msec
229  * interval, we'll start pruning aggressively.
230  */
231 int ip_min_frag_prune_time = 0;

233 static ipft_t   ip_ioctl_ftbl[] = {
234         { IP_IOC_IRE_DELETE, ip_ire_delete, sizeof (ipid_t), 0 },
235         { IP_IOC_IRE_DELETE_NO_REPLY, ip_ire_delete, sizeof (ipid_t),
236                 IPFT_F_NO_REPLY },
237         { IP_IOC_RTS_REQUEST, ip_rts_request, 0, IPFT_F_SELF_REPLY },
238         { 0 }
239 };

241 /* Simple ICMP IP Header Template */
242 static ipha_t icmp_ipha = {
243         IP_SIMPLE_HDR_VERSION, 0, 0, 0, 0, 0, IPPROTO_ICMP
244 };

246 static uchar_t  ip_six_byte_all_ones[] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };

248 static ip_m_t   ip_m_tbl[] = {
249         { DL_ETHER, IFT_ETHER, ETHERTYPE_IP, ETHERTYPE_IPV6,
250             ip_ether_v4_mapping, ip_ether_v6_mapping, ip_ether_v6intfid,
251             ip_nodef_v6intfid },
252         { DL_CSMACD, IFT_ISO88023, ETHERTYPE_IP, ETHERTYPE_IPV6,
253             ip_ether_v4_mapping, ip_ether_v6_mapping, ip_nodef_v6intfid,
254             ip_nodef_v6intfid },
255         { DL_TPB, IFT_ISO88024, ETHERTYPE_IP, ETHERTYPE_IPV6,
256             ip_ether_v4_mapping, ip_ether_v6_mapping, ip_nodef_v6intfid,
257             ip_nodef_v6intfid },
258         { DL_TPR, IFT_ISO88025, ETHERTYPE_IP, ETHERTYPE_IPV6,
259             ip_ether_v4_mapping, ip_ether_v6_mapping, ip_nodef_v6intfid,
```

```
260             ip_nodef_v6intfid },
261         { DL_FDDI, IFT_FDDI, ETHERTYPE_IP, ETHERTYPE_IPV6,
262             ip_ether_v4_mapping, ip_ether_v6_mapping, ip_ether_v6intfid,
263             ip_nodef_v6intfid },
264         { DL_IB, IFT_IB, ETHERTYPE_IP, ETHERTYPE_IPV6,
265             ip_ib_v4_mapping, ip_ib_v6_mapping, ip_ib_v6intfid,
266             ip_nodef_v6intfid },
267         { DL_IPV4, IFT_IPV4, IPPROTO_ENCAP, IPPROTO_IPV6,
268             ip_mbcast_mapping, ip_mbcast_mapping, ip_ipv4_v6intfid,
269             ip_ipv4_v6destintfid },
270         { DL_IPV6, IFT_IPV6, IPPROTO_ENCAP, IPPROTO_IPV6,
271             ip_mbcast_mapping, ip_mbcast_mapping, ip_ipv6_v6intfid,
272             ip_ipv6_v6destintfid },
273         { DL_6TO4, IFT_6TO4, IPPROTO_ENCAP, IPPROTO_IPV6,
274             ip_mbcast_mapping, ip_mbcast_mapping, ip_ipv4_v6intfid,
275             ip_nodef_v6intfid },
276         { SUNW_DL_VNI, IFT_OTHER, ETHERTYPE_IP, ETHERTYPE_IPV6,
277             NULL, NULL, ip_nodef_v6intfid, ip_nodef_v6intfid },
278         { SUNW_DL_IPMP, IFT_OTHER, ETHERTYPE_IP, ETHERTYPE_IPV6,
279             NULL, NULL, ip_ipmp_v6intfid, ip_nodef_v6intfid },
280         { DL_OTHER, IFT_OTHER, ETHERTYPE_IP, ETHERTYPE_IPV6,
281             ip_ether_v4_mapping, ip_ether_v6_mapping, ip_nodef_v6intfid,
282             ip_nodef_v6intfid }
283 };

285 static ill_t    ill_null;               /* Empty ILL for init. */
286 char    ipif_loopback_name[] = "lo0";

288 /* These are used by all IP network modules. */
289 sin6_t  sin6_null;      /* Zero address for quick clears */
290 sin_t   sin_null;       /* Zero address for quick clears */

292 /* When set search for unused ipif_seqid */
293 static ipif_t   ipif_zero;

295 /*
296  * ppa arena is created after these many
297  * interfaces have been plumbed.
298  */
299 uint_t  ill_no_arena = 12;      /* Setable in /etc/system */

301 /*
302  * Allocate per-interface mibs.
303  * Returns true if ok. False otherwise.
304  *  ipsq  may not yet be allocated (loopback case ).
305  */
306 static boolean_t
307 ill_allocate_mibs(ill_t *ill)
308 {
309         /* Already allocated? */
310         if (ill->ill_ip_mib != NULL) {
311                 if (ill->ill_isv6)
312                         ASSERT(ill->ill_icmp6_mib != NULL);
313                 return (B_TRUE);
314         }

316         ill->ill_ip_mib = kmem_zalloc(sizeof (*ill->ill_ip_mib),
317             KM_NOSLEEP);
318         if (ill->ill_ip_mib == NULL) {
319                 return (B_FALSE);
320         }

322         /* Setup static information */
323         SET_MIB(ill->ill_ip_mib->ipIfStatsEntrySize,
324             sizeof (mib2_ipIfStatsEntry_t));
325         if (ill->ill_isv6) {
```



```
326                 ill->ill_ip_mib->ipIfStatsIPVersion = MIB2_INETADDRESSTYPE_ipv6;
327                 SET_MIB(ill->ill_ip_mib->ipIfStatsAddrEntrySize,
328                     sizeof (mib2_ipv6AddrEntry_t));
329                 SET_MIB(ill->ill_ip_mib->ipIfStatsRouteEntrySize,
330                     sizeof (mib2_ipv6RouteEntry_t));
331                 SET_MIB(ill->ill_ip_mib->ipIfStatsNetToMediaEntrySize,
332                     sizeof (mib2_ipv6NetToMediaEntry_t));
333                 SET_MIB(ill->ill_ip_mib->ipIfStatsMemberEntrySize,
334                     sizeof (ipv6_member_t));
335                 SET_MIB(ill->ill_ip_mib->ipIfStatsGroupSourceEntrySize,
336                     sizeof (ipv6_grpsrc_t));
337         } else {
338                 ill->ill_ip_mib->ipIfStatsIPVersion = MIB2_INETADDRESSTYPE_ipv4;
339                 SET_MIB(ill->ill_ip_mib->ipIfStatsAddrEntrySize,
340                     sizeof (mib2_ipAddrEntry_t));
341                 SET_MIB(ill->ill_ip_mib->ipIfStatsRouteEntrySize,
342                     sizeof (mib2_ipRouteEntry_t));
343                 SET_MIB(ill->ill_ip_mib->ipIfStatsNetToMediaEntrySize,
344                     sizeof (mib2_ipNetToMediaEntry_t));
345                 SET_MIB(ill->ill_ip_mib->ipIfStatsMemberEntrySize,
346                     sizeof (ip_member_t));
347                 SET_MIB(ill->ill_ip_mib->ipIfStatsGroupSourceEntrySize,
348                     sizeof (ip_grpsrc_t));

350                 /*
351                  * For a v4 ill, we are done at this point, because per ill
352                  * icmp mibs are only used for v6.
353                  */
354                 return (B_TRUE);
355         }

357         ill->ill_icmp6_mib = kmem_zalloc(sizeof (*ill->ill_icmp6_mib),
358             KM_NOSLEEP);
359         if (ill->ill_icmp6_mib == NULL) {
360                 kmem_free(ill->ill_ip_mib, sizeof (*ill->ill_ip_mib));
361                 ill->ill_ip_mib = NULL;
362                 return (B_FALSE);
363         }
364         /* static icmp info */
365         ill->ill_icmp6_mib->ipv6IfIcmpEntrySize =
366             sizeof (mib2_ipv6IfIcmpEntry_t);
367         /*
368          * The ipIfStatsIfindex and ipv6IfIcmpIndex will be assigned later
369          * after the phyint merge occurs in ipif_set_values -> ill_glist_insert
370          * -> ill_phyint_reinit
371          */
372         return (B_TRUE);
373 }

375 /*
376  * Completely vaporize a lower level tap and all associated interfaces.
377  * ill_delete is called only out of ip_close when the device control
378  * stream is being closed.
379  */
380 void
381 ill_delete(ill_t *ill)
382 {
383         ipif_t  *ipif;
384         ill_t   *prev_ill;
385         ip_stack_t      *ipst = ill->ill_ipst;

387         /*
388          * ill_delete may be forcibly entering the ipsq. The previous
389          * ioctl may not have completed and may need to be aborted.
390          * ipsq_flush takes care of it. If we don't need to enter the
391          * the ipsq forcibly, the 2nd invocation of ipsq_flush in
```

```
392          * ill_delete_tail is sufficient.
393          */
394         ipsq_flush(ill);

396         /*
397          * Nuke all interfaces.  ipif_free will take down the interface,
398          * remove it from the list, and free the data structure.
399          * Walk down the ipif list and remove the logical interfaces
400          * first before removing the main ipif. We can't unplumb
401          * zeroth interface first in the case of IPv6 as update_conn_ill
402          * -> ip_ll_multireq de-references ill_ipif for checking
403          * POINTOPOINT.
404          *
405          * If ill_ipif was not properly initialized (i.e low on memory),
406          * then no interfaces to clean up. In this case just clean up the
407          * ill.
408          */
409         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next)
410                 ipif_free(ipif);

412         /*
413          * clean out all the nce_t entries that depend on this
414          * ill for the ill_phys_addr.
415          */
416         nce_flush(ill, B_TRUE);

418         /* Clean up msgs on pending upcalls for mrouted */
419         reset_mrt_ill(ill);

421         update_conn_ill(ill, ipst);

423         /*
424          * Remove multicast references added as a result of calls to
425          * ip_join_allmulti().
426          */
427         ip_purge_allmulti(ill);

429         /*
430          * If the ill being deleted is under IPMP, boot it out of the illgrp.
431          */
432         if (IS_UNDER_IPMP(ill))
433                 ipmp_ill_leave_illgrp(ill);

435         /*
436          * ill_down will arrange to blow off any IRE's dependent on this
437          * ILL, and shut down fragmentation reassembly.
438          */
439         ill_down(ill);

441         /* Let SCTP know, so that it can remove this from its list. */
442         sctp_update_ill(ill, SCTP_ILL_REMOVE);

444         /*
445          * Walk all CONNs that can have a reference on an ire or nce for this
446          * ill (we actually walk all that now have stale references).
447          */
448         ipcl_walk(conn_ixa_cleanup, (void *)B_TRUE, ipst);

450         /* With IPv6 we have dce_ifindex. Cleanup for neatness */
451         if (ill->ill_isv6)
452                 dce_cleanup(ill->ill_phyint->phyint_ifindex, ipst);

454         /*
455          * If an address on this ILL is being used as a source address then
456          * clear out the pointers in other ILLs that point to this ILL.
457          */
```



```
458         rw_enter(&ipst->ips_ill_g_usesrc_lock, RW_WRITER);
459         if (ill->ill_usesrc_grp_next != NULL) {
460                 if (ill->ill_usesrc_ifindex == 0) { /* usesrc ILL ? */
461                         ill_disband_usesrc_group(ill);
462                 } else {        /* consumer of the usesrc ILL */
463                         prev_ill = ill_prev_usesrc(ill);
464                         prev_ill->ill_usesrc_grp_next =
465                             ill->ill_usesrc_grp_next;
466                 }
467         }
468         rw_exit(&ipst->ips_ill_g_usesrc_lock);
469 }

471 static void
472 ipif_non_duplicate(ipif_t *ipif)
473 {
474         ill_t *ill = ipif->ipif_ill;
475         mutex_enter(&ill->ill_lock);
476         if (ipif->ipif_flags & IPIF_DUPLICATE) {
477                 ipif->ipif_flags &= ~IPIF_DUPLICATE;
478                 ASSERT(ill->ill_ipif_dup_count > 0);
479                 ill->ill_ipif_dup_count--;
480         }
481         mutex_exit(&ill->ill_lock);
482 }

484 /*
485  * ill_delete_tail is called from ip_modclose after all references
486  * to the closing ill are gone. The wait is done in ip_modclose
487  */
488 void
489 ill_delete_tail(ill_t *ill)
490 {
491         mblk_t  **mpp;
492         ipif_t  *ipif;
493         ip_stack_t *ipst = ill->ill_ipst;

495         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
496                 ipif_non_duplicate(ipif);
497                 (void) ipif_down_tail(ipif);
498         }

500         ASSERT(ill->ill_ipif_dup_count == 0);

502         /*
503          * If polling capability is enabled (which signifies direct
504          * upcall into IP and driver has ill saved as a handle),
505          * we need to make sure that unbind has completed before we
506          * let the ill disappear and driver no longer has any reference
507          * to this ill.
508          */
509         mutex_enter(&ill->ill_lock);
510         while (ill->ill_state_flags & ILL_DL_UNBIND_IN_PROGRESS)
511                 cv_wait(&ill->ill_cv, &ill->ill_lock);
512         mutex_exit(&ill->ill_lock);
513         ASSERT(!(ill->ill_capabilities &
514             (ILL_CAPAB_DLD | ILL_CAPAB_DLD_POLL | ILL_CAPAB_DLD_DIRECT)));

516         if (ill->ill_net_type != IRE_LOOPBACK)
517                 qprocsoff(ill->ill_rq);

519         /*
520          * We do an ipsq_flush once again now. New messages could have
521          * landed up from below (M_ERROR or M_HANGUP). Similarly ioctls
522          * could also have landed up if an ioctl thread had looked up
523          * the ill before we set the ILL_CONDEMNED flag, but not yet
```

```
524          * enqueued the ioctl when we did the ipsq_flush last time.
525          */
526         ipsq_flush(ill);

528         /*
529          * Free capabilities.
530          */
531         if (ill->ill_hcksum_capab != NULL) {
532                 kmem_free(ill->ill_hcksum_capab, sizeof (ill_hcksum_capab_t));
533                 ill->ill_hcksum_capab = NULL;
534         }

536         if (ill->ill_zerocopy_capab != NULL) {
537                 kmem_free(ill->ill_zerocopy_capab,
538                     sizeof (ill_zerocopy_capab_t));
539                 ill->ill_zerocopy_capab = NULL;
540         }

542         if (ill->ill_lso_capab != NULL) {
543                 kmem_free(ill->ill_lso_capab, sizeof (ill_lso_capab_t));
544                 ill->ill_lso_capab = NULL;
545         }

547         if (ill->ill_dld_capab != NULL) {
548                 kmem_free(ill->ill_dld_capab, sizeof (ill_dld_capab_t));
549                 ill->ill_dld_capab = NULL;
550         }

552         /* Clean up ill_allowed_ips* related state */
553         if (ill->ill_allowed_ips != NULL) {
554                 ASSERT(ill->ill_allowed_ips_cnt > 0);
555                 kmem_free(ill->ill_allowed_ips,
556                     ill->ill_allowed_ips_cnt * sizeof (in6_addr_t));
557                 ill->ill_allowed_ips = NULL;
558                 ill->ill_allowed_ips_cnt = 0;
559         }

561         while (ill->ill_ipif != NULL)
562                 ipif_free_tail(ill->ill_ipif);

564         /*
565          * We have removed all references to ilm from conn and the ones joined
566          * within the kernel.
567          *
568          * We don't walk conns, mrts and ires because
569          *
570          * 1) update_conn_ill and reset_mrt_ill cleans up conns and mrts.
571          * 2) ill_down ->ill_downi walks all the ires and cleans up
572          *    ill references.
573          */

575         /*
576          * If this ill is an IPMP meta-interface, blow away the illgrp.  This
577          * is safe to do because the illgrp has already been unlinked from the
578          * group by I_PUNLINK, and thus SIOCSLIFGROUPNAME cannot find it.
579          */
580         if (IS_IPMP(ill)) {
581                 ipmp_illgrp_destroy(ill->ill_grp);
582                 ill->ill_grp = NULL;
583         }

585         if (ill->ill_mphysaddr_list != NULL) {
586                 multiphysaddr_t *mpa, *tmpa;

588                 mpa = ill->ill_mphysaddr_list;
589                 ill->ill_mphysaddr_list = NULL;
```



```
590                 while (mpa) {
591                         tmpa = mpa->mpa_next;
592                         kmem_free(mpa, sizeof (*mpa));
593                         mpa = tmpa;
594                 }
595         }
596         /*
597          * Take us out of the list of ILLs. ill_glist_delete -> phyint_free
598          * could free the phyint. No more reference to the phyint after this
599          * point.
600          */
601         (void) ill_glist_delete(ill);

603         if (ill->ill_frag_ptr != NULL) {
604                 uint_t count;

606                 for (count = 0; count < ILL_FRAG_HASH_TBL_COUNT; count++) {
607                         mutex_destroy(&ill->ill_frag_hash_tbl[count].ipfb_lock);
608                 }
609                 mi_free(ill->ill_frag_ptr);
610                 ill->ill_frag_ptr = NULL;
611                 ill->ill_frag_hash_tbl = NULL;
612         }

614         freemsg(ill->ill_nd_lla_mp);
615         /* Free all retained control messages. */
616         mpp = &ill->ill_first_mp_to_free;
617         do {
618                 while (mpp[0]) {
619                         mblk_t  *mp;
620                         mblk_t  *mp1;

622                         mp = mpp[0];
623                         mpp[0] = mp->b_next;
624                         for (mp1 = mp; mp1 != NULL; mp1 = mp1->b_cont) {
625                                 mp1->b_next = NULL;
626                                 mp1->b_prev = NULL;
627                         }
628                         freemsg(mp);
629                 }
630         } while (mpp++ != &ill->ill_last_mp_to_free);

632         ill_free_mib(ill);

634 #ifdef DEBUG
635         ill_trace_cleanup(ill);
636 #endif

638         /* The default multicast interface might have changed */
639         ire_increment_multicast_generation(ipst, ill->ill_isv6);

641         /* Drop refcnt here */
642         netstack_rele(ill->ill_ipst->ips_netstack);
643         ill->ill_ipst = NULL;
644 }

646 static void
647 ill_free_mib(ill_t *ill)
648 {
649         ip_stack_t *ipst = ill->ill_ipst;

651         /*
652          * MIB statistics must not be lost, so when an interface
653          * goes away the counter values will be added to the global
654          * MIBs.
655          */
```

```
656         if (ill->ill_ip_mib != NULL) {
657                 if (ill->ill_isv6) {
658                         ip_mib2_add_ip_stats(&ipst->ips_ip6_mib,
659                             ill->ill_ip_mib);
660                 } else {
661                         ip_mib2_add_ip_stats(&ipst->ips_ip_mib,
662                             ill->ill_ip_mib);
663                 }

665                 kmem_free(ill->ill_ip_mib, sizeof (*ill->ill_ip_mib));
666                 ill->ill_ip_mib = NULL;
667         }
668         if (ill->ill_icmp6_mib != NULL) {
669                 ip_mib2_add_icmp6_stats(&ipst->ips_icmp6_mib,
670                     ill->ill_icmp6_mib);
671                 kmem_free(ill->ill_icmp6_mib, sizeof (*ill->ill_icmp6_mib));
672                 ill->ill_icmp6_mib = NULL;
673         }
674 }

676 /*
677  * Concatenate together a physical address and a sap.
678  *
679  * Sap_lengths are interpreted as follows:
680  *   sap_length == 0    ==>     no sap
681  *   sap_length > 0     ==>     sap is at the head of the dlpi address
682  *   sap_length < 0     ==>     sap is at the tail of the dlpi address
683  */
684 static void
685 ill_dlur_copy_address(uchar_t *phys_src, uint_t phys_length,
686     t_scalar_t sap_src, t_scalar_t sap_length, uchar_t *dst)
687 {
688         uint16_t sap_addr = (uint16_t)sap_src;

690         if (sap_length == 0) {
691                 if (phys_src == NULL)
692                         bzero(dst, phys_length);
693                 else
694                         bcopy(phys_src, dst, phys_length);
695         } else if (sap_length < 0) {
696                 if (phys_src == NULL)
697                         bzero(dst, phys_length);
698                 else
699                         bcopy(phys_src, dst, phys_length);
700                 bcopy(&sap_addr, (char *)dst + phys_length, sizeof (sap_addr));
701         } else {
702                 bcopy(&sap_addr, dst, sizeof (sap_addr));
703                 if (phys_src == NULL)
704                         bzero((char *)dst + sap_length, phys_length);
705                 else
706                         bcopy(phys_src, (char *)dst + sap_length, phys_length);
707         }
708 }

710 /*
711  * Generate a dl_unitdata_req mblk for the device and address given.
712  * addr_length is the length of the physical portion of the address.
713  * If addr is NULL include an all zero address of the specified length.
714  * TRUE? In any case, addr_length is taken to be the entire length of the
715  * dlpi address, including the absolute value of sap_length.
716  */
717 mblk_t *
718 ill_dlur_gen(uchar_t *addr, uint_t addr_length, t_uscalar_t sap,
719                 t_scalar_t sap_length)
720 {
721         dl_unitdata_req_t *dlur;
```



```
722         mblk_t  *mp;
723         t_scalar_t      abs_sap_length;         /* absolute value */

725         abs_sap_length = ABS(sap_length);
726         mp = ip_dlpi_alloc(sizeof (*dlur) + addr_length + abs_sap_length,
727             DL_UNITDATA_REQ);
728         if (mp == NULL)
729                 return (NULL);
730         dlur = (dl_unitdata_req_t *)mp->b_rptr;
731         /* HACK: accomodate incompatible DLPI drivers */
732         if (addr_length == 8)
733                 addr_length = 6;
734         dlur->dl_dest_addr_length = addr_length + abs_sap_length;
735         dlur->dl_dest_addr_offset = sizeof (*dlur);
736         dlur->dl_priority.dl_min = 0;
737         dlur->dl_priority.dl_max = 0;
738         ill_dlur_copy_address(addr, addr_length, sap, sap_length,
739             (uchar_t *)&dlur[1]);
740         return (mp);
741 }

743 /*
744  * Add the pending mp to the list. There can be only 1 pending mp
745  * in the list. Any exclusive ioctl that needs to wait for a response
746  * from another module or driver needs to use this function to set
747  * the ipx_pending_mp to the ioctl mblk and wait for the response from
748  * the other module/driver. This is also used while waiting for the
749  * ipif/ill/ire refcnts to drop to zero in bringing down an ipif.
750  */
751 boolean_t
752 ipsq_pending_mp_add(conn_t *connp, ipif_t *ipif, queue_t *q, mblk_t *add_mp,
753     int waitfor)
754 {
755         ipxop_t *ipx = ipif->ipif_ill->ill_phyint->phyint_ipsq->ipsq_xop;

757         ASSERT(IAM_WRITER_IPIF(ipif));
758         ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));
759         ASSERT((add_mp->b_next == NULL) && (add_mp->b_prev == NULL));
760         ASSERT(ipx->ipx_pending_mp == NULL);
761         /*
762          * The caller may be using a different ipif than the one passed into
763          * ipsq_current_start() (e.g., suppose an ioctl that came in on the V4
764          * ill needs to wait for the V6 ill to quiesce).  So we can't ASSERT
765          * that 'ipx_current_ipif == ipif'.
766          */
767         ASSERT(ipx->ipx_current_ipif != NULL);

769         /*
770          * M_IOCDATA from ioctls, M_ERROR/M_HANGUP/M_PROTO/M_PCPROTO from the
771          * driver.
772          */
773         ASSERT((DB_TYPE(add_mp) == M_IOCDATA) || (DB_TYPE(add_mp) == M_ERROR) ||
774             (DB_TYPE(add_mp) == M_HANGUP) || (DB_TYPE(add_mp) == M_PROTO) ||
775             (DB_TYPE(add_mp) == M_PCPROTO));

777         if (connp != NULL) {
778                 ASSERT(MUTEX_HELD(&connp->conn_lock));
779                 /*
780                  * Return error if the conn has started closing. The conn
781                  * could have finished cleaning up the pending mp list,
782                  * If so we should not add another mp to the list negating
783                  * the cleanup.
784                  */
785                 if (connp->conn_state_flags & CONN_CLOSING)
786                         return (B_FALSE);
787         }
```

```
788         mutex_enter(&ipx->ipx_lock);
789         ipx->ipx_pending_ipif = ipif;
790         /*
791          * Note down the queue in b_queue. This will be returned by
792          * ipsq_pending_mp_get. Caller will then use these values to restart
793          * the processing
794          */
795         add_mp->b_next = NULL;
796         add_mp->b_queue = q;
797         ipx->ipx_pending_mp = add_mp;
798         ipx->ipx_waitfor = waitfor;
799         mutex_exit(&ipx->ipx_lock);

801         if (connp != NULL)
802                 connp->conn_oper_pending_ill = ipif->ipif_ill;

804         return (B_TRUE);
805 }

807 /*
808  * Retrieve the ipx_pending_mp and return it. There can be only 1 mp
809  * queued in the list.
810  */
811 mblk_t *
812 ipsq_pending_mp_get(ipsq_t *ipsq, conn_t **connpp)
813 {
814         mblk_t  *curr = NULL;
815         ipxop_t *ipx = ipsq->ipsq_xop;

817         *connpp = NULL;
818         mutex_enter(&ipx->ipx_lock);
819         if (ipx->ipx_pending_mp == NULL) {
820                 mutex_exit(&ipx->ipx_lock);
821                 return (NULL);
822         }

824         /* There can be only 1 such excl message */
825         curr = ipx->ipx_pending_mp;
826         ASSERT(curr->b_next == NULL);
827         ipx->ipx_pending_ipif = NULL;
828         ipx->ipx_pending_mp = NULL;
829         ipx->ipx_waitfor = 0;
830         mutex_exit(&ipx->ipx_lock);

832         if (CONN_Q(curr->b_queue)) {
833                 /*
834                  * This mp did a refhold on the conn, at the start of the ioctl.
835                  * So we can safely return a pointer to the conn to the caller.
836                  */
837                 *connpp = Q_TO_CONN(curr->b_queue);
838         } else {
839                 *connpp = NULL;
840         }
841         curr->b_next = NULL;
842         curr->b_prev = NULL;
843         return (curr);
844 }

846 /*
847  * Cleanup the ioctl mp queued in ipx_pending_mp
848  * - Called in the ill_delete path
849  * - Called in the M_ERROR or M_HANGUP path on the ill.
850  * - Called in the conn close path.
851  *
852  * Returns success on finding the pending mblk associated with the ioctl or
853  * exclusive operation in progress, failure otherwise.
```



```
854  */
855 boolean_t
856 ipsq_pending_mp_cleanup(ill_t *ill, conn_t *connp)
857 {
858         mblk_t  *mp;
859         ipxop_t *ipx;
860         queue_t *q;
861         ipif_t  *ipif;
862         int     cmd;

864         ASSERT(IAM_WRITER_ILL(ill));
865         ipx = ill->ill_phyint->phyint_ipsq->ipsq_xop;

867         mutex_enter(&ipx->ipx_lock);
868         mp = ipx->ipx_pending_mp;
869         if (connp != NULL) {
870                 if (mp == NULL || mp->b_queue != CONNP_TO_WQ(connp)) {
871                         /*
872                          * Nothing to clean since the conn that is closing
873                          * does not have a matching pending mblk in
874                          * ipx_pending_mp.
875                          */
876                         mutex_exit(&ipx->ipx_lock);
877                         return (B_FALSE);
878                 }
879         } else {
880                 /*
881                  * A non-zero ill_error signifies we are called in the
882                  * M_ERROR or M_HANGUP path and we need to unconditionally
883                  * abort any current ioctl and do the corresponding cleanup.
884                  * A zero ill_error means we are in the ill_delete path and
885                  * we do the cleanup only if there is a pending mp.
886                  */
887                 if (mp == NULL && ill->ill_error == 0) {
888                         mutex_exit(&ipx->ipx_lock);
889                         return (B_FALSE);
890                 }
891         }

893         /* Now remove from the ipx_pending_mp */
894         ipx->ipx_pending_mp = NULL;
895         ipif = ipx->ipx_pending_ipif;
896         ipx->ipx_pending_ipif = NULL;
897         ipx->ipx_waitfor = 0;
898         ipx->ipx_current_ipif = NULL;
899         cmd = ipx->ipx_current_ioctl;
900         ipx->ipx_current_ioctl = 0;
901         ipx->ipx_current_done = B_TRUE;
902         mutex_exit(&ipx->ipx_lock);

904         if (mp == NULL)
905                 return (B_FALSE);

907         q = mp->b_queue;
908         mp->b_next = NULL;
909         mp->b_prev = NULL;
910         mp->b_queue = NULL;

912         if (DB_TYPE(mp) == M_IOCTL || DB_TYPE(mp) == M_IOCDATA) {
913                 DTRACE_PROBE4(ipif__ioctl,
914                     char *, "ipsq_pending_mp_cleanup",
915                     int, cmd, ill_t *, ipif == NULL ? NULL : ipif->ipif_ill,
916                     ipif_t *, ipif);
917                 if (connp == NULL) {
918                         ip_ioctl_finish(q, mp, ENXIO, NO_COPYOUT, NULL);
919                 } else {
```

```
920                         ip_ioctl_finish(q, mp, ENXIO, CONN_CLOSE, NULL);
921                         mutex_enter(&ipif->ipif_ill->ill_lock);
922                         ipif->ipif_state_flags &= ~IPIF_CHANGING;
923                         mutex_exit(&ipif->ipif_ill->ill_lock);
924                 }
925         } else {
926                 inet_freemsg(mp);
927         }
928         return (B_TRUE);
929 }

931 /*
932  * Called in the conn close path and ill delete path
933  */
934 static void
935 ipsq_xopq_mp_cleanup(ill_t *ill, conn_t *connp)
936 {
937         ipsq_t  *ipsq;
938         mblk_t  *prev;
939         mblk_t  *curr;
940         mblk_t  *next;
941         queue_t *wq, *rq = NULL;
942         mblk_t  *tmp_list = NULL;

944         ASSERT(IAM_WRITER_ILL(ill));
945         if (connp != NULL)
946                 wq = CONNP_TO_WQ(connp);
947         else
948                 wq = ill->ill_wq;

950         /*
951          * In the case of lo0 being unplumbed, ill_wq will be NULL. Guard
952          * against this here.
953          */
954         if (wq != NULL)
955                 rq = RD(wq);

957         ipsq = ill->ill_phyint->phyint_ipsq;
958         /*
959          * Cleanup the ioctl mp's queued in ipsq_xopq_pending_mp if any.
960          * In the case of ioctl from a conn, there can be only 1 mp
961          * queued on the ipsq. If an ill is being unplumbed flush all
962          * the messages.
963          */
964         mutex_enter(&ipsq->ipsq_lock);
965         for (prev = NULL, curr = ipsq->ipsq_xopq_mphead; curr != NULL;
966             curr = next) {
967                 next = curr->b_next;
968                 if (connp == NULL ||
969                     (curr->b_queue == wq || curr->b_queue == rq)) {
970                         /* Unlink the mblk from the pending mp list */
971                         if (prev != NULL) {
972                                 prev->b_next = curr->b_next;
973                         } else {
974                                 ASSERT(ipsq->ipsq_xopq_mphead == curr);
975                                 ipsq->ipsq_xopq_mphead = curr->b_next;
976                         }
977                         if (ipsq->ipsq_xopq_mptail == curr)
978                                 ipsq->ipsq_xopq_mptail = prev;
979                         /*
980                          * Create a temporary list and release the ipsq lock
981                          * New elements are added to the head of the tmp_list
982                          */
983                         curr->b_next = tmp_list;
984                         tmp_list = curr;
985                 } else {
```



```
986                         prev = curr;
987                 }
988         }
989         mutex_exit(&ipsq->ipsq_lock);

991         while (tmp_list != NULL) {
992                 curr = tmp_list;
993                 tmp_list = curr->b_next;
994                 curr->b_next = NULL;
995                 curr->b_prev = NULL;
996                 wq = curr->b_queue;
997                 curr->b_queue = NULL;
998                 if (DB_TYPE(curr) == M_IOCTL || DB_TYPE(curr) == M_IOCDATA) {
999                         DTRACE_PROBE4(ipif__ioctl,
1000                            char *, "ipsq_xopq_mp_cleanup",
1001                            int, 0, ill_t *, NULL, ipif_t *, NULL);
1002                        ip_ioctl_finish(wq, curr, ENXIO, connp != NULL ?
1003                            CONN_CLOSE : NO_COPYOUT, NULL);
1004                } else {
1005                        /*
1006                         * IP-MT XXX In the case of TLI/XTI bind / optmgmt
1007                         * this can't be just inet_freemsg. we have to
1008                         * restart it otherwise the thread will be stuck.
1009                         */
1010                        inet_freemsg(curr);
1011                }
1012        }
1013 }

1015 /*
1016  * This conn has started closing. Cleanup any pending ioctl from this conn.
1017  * STREAMS ensures that there can be at most 1 active ioctl on a stream.
1018  */
1019 void
1020 conn_ioctl_cleanup(conn_t *connp)
1021 {
1022         ipsq_t  *ipsq;
1023         ill_t   *ill;
1024         boolean_t refheld;

1026         /*
1027          * Check for a queued ioctl. If the ioctl has not yet started, the mp
1028          * is pending in the list headed by ipsq_xopq_head. If the ioctl has
1029          * started the mp could be present in ipx_pending_mp. Note that if
1030          * conn_oper_pending_ill is NULL, the ioctl may still be in flight and
1031          * not yet queued anywhere. In this case, the conn close code will wait
1032          * until the conn_ref is dropped. If the stream was a tcp stream, then
1033          * tcp_close will wait first until all ioctls have completed for this
1034          * conn.
1035          */
1036         mutex_enter(&connp->conn_lock);
1037         ill = connp->conn_oper_pending_ill;
1038         if (ill == NULL) {
1039                 mutex_exit(&connp->conn_lock);
1040                 return;
1041         }

1043         /*
1044          * We may not be able to refhold the ill if the ill/ipif
1045          * is changing. But we need to make sure that the ill will
1046          * not vanish. So we just bump up the ill_waiter count.
1047          */
1048         refheld = ill_waiter_inc(ill);
1049         mutex_exit(&connp->conn_lock);
1050         if (refheld) {
1051                 if (ipsq_enter(ill, B_TRUE, NEW_OP)) {
```

```
1052                        ill_waiter_dcr(ill);
1053                        /*
1054                         * Check whether this ioctl has started and is
1055                         * pending. If it is not found there then check
1056                         * whether this ioctl has not even started and is in
1057                         * the ipsq_xopq list.
1058                         */
1059                        if (!ipsq_pending_mp_cleanup(ill, connp))
1060                                ipsq_xopq_mp_cleanup(ill, connp);
1061                        ipsq = ill->ill_phyint->phyint_ipsq;
1062                        ipsq_exit(ipsq);
1063                        return;
1064                }
1065        }

1067        /*
1068         * The ill is also closing and we could not bump up the
1069         * ill_waiter_count or we could not enter the ipsq. Leave
1070         * the cleanup to ill_delete
1071         */
1072        mutex_enter(&connp->conn_lock);
1073        while (connp->conn_oper_pending_ill != NULL)
1074                cv_wait(&connp->conn_refcv, &connp->conn_lock);
1075        mutex_exit(&connp->conn_lock);
1076        if (refheld)
1077                ill_waiter_dcr(ill);
1078 }

1080 /*
1081  * ipcl_walk function for cleaning up conn_*_ill fields.
1082  * Note that we leave ixa_multicast_ifindex, conn_incoming_ifindex, and
1083  * conn_bound_if in place. We prefer dropping
1084  * packets instead of sending them out the wrong interface, or accepting
1085  * packets from the wrong ifindex.
1086  */
1087 static void
1088 conn_cleanup_ill(conn_t *connp, caddr_t arg)
1089 {
1090        ill_t   *ill = (ill_t *)arg;

1092        mutex_enter(&connp->conn_lock);
1093        if (connp->conn_dhcpinit_ill == ill) {
1094                connp->conn_dhcpinit_ill = NULL;
1095                ASSERT(ill->ill_dhcpinit != 0);
1096                atomic_dec_32(&ill->ill_dhcpinit);
1097                ill_set_inputfn(ill);
1098        }
1099        mutex_exit(&connp->conn_lock);
1100 }

1102 static int
1103 ill_down_ipifs_tail(ill_t *ill)
1104 {
1105        ipif_t  *ipif;
1106        int err;

1108        ASSERT(IAM_WRITER_ILL(ill));
1109        for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
1110                ipif_non_duplicate(ipif);
1111                /*
1112                 * ipif_down_tail will call arp_ll_down on the last ipif
1113                 * and typically return EINPROGRESS when the DL_UNBIND is sent.
1114                 */
1115                if ((err = ipif_down_tail(ipif)) != 0)
1116                        return (err);
1117        }
```



```
1118        return (0);
1119 }

1121 /* ARGSUSED */
1122 void
1123 ipif_all_down_tail(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
1124 {
1125        ASSERT(IAM_WRITER_IPSQ(ipsq));
1126        (void) ill_down_ipifs_tail(q->q_ptr);
1127        freemsg(mp);
1128        ipsq_current_finish(ipsq);
1129 }

1131 /*
1132  * ill_down_start is called when we want to down this ill and bring it up again
1133  * It is called when we receive an M_ERROR / M_HANGUP. In this case we shut down
1134  * all interfaces, but don't tear down any plumbing.
1135  */
1136 boolean_t
1137 ill_down_start(queue_t *q, mblk_t *mp)
1138 {
1139        ill_t   *ill = q->q_ptr;
1140        ipif_t  *ipif;

1142        ASSERT(IAM_WRITER_ILL(ill));
1143        /*
1144         * It is possible that some ioctl is already in progress while we
1145         * received the M_ERROR / M_HANGUP in which case, we need to abort
1146         * the ioctl. ill_down_start() is being processed as CUR_OP rather
1147         * than as NEW_OP since the cause of the M_ERROR / M_HANGUP may prevent
1148         * the in progress ioctl from ever completing.
1149         *
1150         * The thread that started the ioctl (if any) must have returned,
1151         * since we are now executing as writer. After the 2 calls below,
1152         * the state of the ipsq and the ill would reflect no trace of any
1153         * pending operation. Subsequently if there is any response to the
1154         * original ioctl from the driver, it would be discarded as an
1155         * unsolicited message from the driver.
1156         */
1157        (void) ipsq_pending_mp_cleanup(ill, NULL);
1158        ill_dlpi_clear_deferred(ill);

1160        for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next)
1161                (void) ipif_down(ipif, NULL, NULL);

1163        ill_down(ill);

1165        /*
1166         * Walk all CONNs that can have a reference on an ire or nce for this
1167         * ill (we actually walk all that now have stale references).
1168         */
1169        ipcl_walk(conn_ixa_cleanup, (void *)B_TRUE, ill->ill_ipst);

1171        /* With IPv6 we have dce_ifindex. Cleanup for neatness */
1172        if (ill->ill_isv6)
1173                dce_cleanup(ill->ill_phyint->phyint_ifindex, ill->ill_ipst);

1175        ipsq_current_start(ill->ill_phyint->phyint_ipsq, ill->ill_ipif, 0);

1177        /*
1178         * Atomically test and add the pending mp if references are active.
1179         */
1180        mutex_enter(&ill->ill_lock);
1181        if (!ill_is_quiescent(ill)) {
1182                /* call cannot fail since 'conn_t *' argument is NULL */
1183                (void) ipsq_pending_mp_add(NULL, ill->ill_ipif, ill->ill_rq,
```

```
1184                    mp, ILL_DOWN);
1185                mutex_exit(&ill->ill_lock);
1186                return (B_FALSE);
1187        }
1188        mutex_exit(&ill->ill_lock);
1189        return (B_TRUE);
1190 }

1192 static void
1193 ill_down(ill_t *ill)
1194 {
1195        mblk_t  *mp;
1196        ip_stack_t      *ipst = ill->ill_ipst;

1198        /*
1199         * Blow off any IREs dependent on this ILL.
1200         * The caller needs to handle conn_ixa_cleanup
1201         */
1202        ill_delete_ires(ill);

1204        ire_walk_ill(0, 0, ill_downi, ill, ill);

1206        /* Remove any conn_*_ill depending on this ill */
1207        ipcl_walk(conn_cleanup_ill, (caddr_t)ill, ipst);

1209        /*
1210         * Free state for additional IREs.
1211         */
1212        mutex_enter(&ill->ill_saved_ire_lock);
1213        mp = ill->ill_saved_ire_mp;
1214        ill->ill_saved_ire_mp = NULL;
1215        ill->ill_saved_ire_cnt = 0;
1216        mutex_exit(&ill->ill_saved_ire_lock);
1217        freemsg(mp);
1218 }

1220 /*
1221  * ire_walk routine used to delete every IRE that depends on
1222  * 'ill'.  (Always called as writer, and may only be called from ire_walk.)
1223  *
1224  * Note: since the routes added by the kernel are deleted separately,
1225  * this will only be 1) IRE_IF_CLONE and 2) manually added IRE_INTERFACE.
1226  *
1227  * We also remove references on ire_nce_cache entries that refer to the ill.
1228  */
1229 void
1230 ill_downi(ire_t *ire, char *ill_arg)
1231 {
1232        ill_t   *ill = (ill_t *)ill_arg;
1233        nce_t   *nce;

1235        mutex_enter(&ire->ire_lock);
1236        nce = ire->ire_nce_cache;
1237        if (nce != NULL && nce->nce_ill == ill)
1238                ire->ire_nce_cache = NULL;
1239        else
1240                nce = NULL;
1241        mutex_exit(&ire->ire_lock);
1242        if (nce != NULL)
1243                nce_refrele(nce);
1244        if (ire->ire_ill == ill) {
1245                /*
1246                 * The existing interface binding for ire must be
1247                 * deleted before trying to bind the route to another
1248                 * interface. However, since we are using the contents of the
1249                 * ire after ire_delete, the caller has to ensure that
```



```
1250                 * CONDEMNED (deleted) ire's are not removed from the list
1251                 * when ire_delete() returns. Currently ill_downi() is
1252                 * only called as part of ire_walk*() routines, so that
1253                 * the irb_refhold() done by ire_walk*() will ensure that
1254                 * ire_delete() does not lead to ire_inactive().
1255                 */
1256                ASSERT(ire->ire_bucket->irb_refcnt > 0);
1257                ire_delete(ire);
1258                if (ire->ire_unbound)
1259                        ire_rebind(ire);
1260        }
1261 }

1263 /* Remove IRE_IF_CLONE on this ill */
1264 void
1265 ill_downi_if_clone(ire_t *ire, char *ill_arg)
1266 {
1267        ill_t   *ill = (ill_t *)ill_arg;

1269        ASSERT(ire->ire_type & IRE_IF_CLONE);
1270        if (ire->ire_ill == ill)
1271                ire_delete(ire);
1272 }

1274 /* Consume an M_IOCACK of the fastpath probe. */
1275 void
1276 ill_fastpath_ack(ill_t *ill, mblk_t *mp)
1277 {
1278        mblk_t  *mp1 = mp;

1280        /*
1281         * If this was the first attempt turn on the fastpath probing.
1282         */
1283        mutex_enter(&ill->ill_lock);
1284        if (ill->ill_dlpi_fastpath_state == IDS_INPROGRESS)
1285                ill->ill_dlpi_fastpath_state = IDS_OK;
1286        mutex_exit(&ill->ill_lock);

1288        /* Free the M_IOCACK mblk, hold on to the data */
1289        mp = mp->b_cont;
1290        freeb(mp1);
1291        if (mp == NULL)
1292                return;
1293        if (mp->b_cont != NULL)
1294                nce_fastpath_update(ill, mp);
1295        else
1296                ip0dbg(("ill_fastpath_ack:  no b_cont\n"));
1297        freemsg(mp);
1298 }

1300 /*
1301  * Throw an M_IOCTL message downstream asking "do you know fastpath?"
1302  * The data portion of the request is a dl_unitdata_req_t template for
1303  * what we would send downstream in the absence of a fastpath confirmation.
1304  */
1305 int
1306 ill_fastpath_probe(ill_t *ill, mblk_t *dlur_mp)
1307 {
1308        struct iocblk   *ioc;
1309        mblk_t  *mp;

1311        if (dlur_mp == NULL)
1312                return (EINVAL);

1314        mutex_enter(&ill->ill_lock);
1315        switch (ill->ill_dlpi_fastpath_state) {
```

```
1316         case IDS_FAILED:
1317                 /*
1318                  * Driver NAKed the first fastpath ioctl - assume it doesn't
1319                  * support it.
1320                  */
1321                 mutex_exit(&ill->ill_lock);
1322                 return (ENOTSUP);
1323         case IDS_UNKNOWN:
1324                 /* This is the first probe */
1325                 ill->ill_dlpi_fastpath_state = IDS_INPROGRESS;
1326                 break;
1327         default:
1328                 break;
1329         }
1330         mutex_exit(&ill->ill_lock);

1332         if ((mp = mkiocb(DL_IOC_HDR_INFO)) == NULL)
1333                 return (EAGAIN);

1335         mp->b_cont = copyb(dlur_mp);
1336         if (mp->b_cont == NULL) {
1337                 freeb(mp);
1338                 return (EAGAIN);
1339         }

1341         ioc = (struct iocblk *)mp->b_rptr;
1342         ioc->ioc_count = msgdsize(mp->b_cont);

1344         DTRACE_PROBE3(ill__dlpi, char *, "ill_fastpath_probe",
1345             char *, "DL_IOC_HDR_INFO", ill_t *, ill);
1346         putnext(ill->ill_wq, mp);
1347         return (0);
1348 }

1350 void
1351 ill_capability_probe(ill_t *ill)
1352 {
1353         mblk_t  *mp;

1355         ASSERT(IAM_WRITER_ILL(ill));

1357         if (ill->ill_dlpi_capab_state != IDCS_UNKNOWN &&
1358             ill->ill_dlpi_capab_state != IDCS_FAILED)
1359                 return;

1361         /*
1362          * We are starting a new cycle of capability negotiation.
1363          * Free up the capab reset messages of any previous incarnation.
1364          * We will do a fresh allocation when we get the response to our probe
1365          */
1366         if (ill->ill_capab_reset_mp != NULL) {
1367                 freemsg(ill->ill_capab_reset_mp);
1368                 ill->ill_capab_reset_mp = NULL;
1369         }

1371         ip1dbg(("ill_capability_probe: starting capability negotiation\n"));

1373         mp = ip_dlpi_alloc(sizeof (dl_capability_req_t), DL_CAPABILITY_REQ);
1374         if (mp == NULL)
1375                 return;

1377         ill_capability_send(ill, mp);
1378         ill->ill_dlpi_capab_state = IDCS_PROBE_SENT;
1379 }

1381 void
```



```
1382 ill_capability_reset(ill_t *ill, boolean_t reneg)
1383 {
1384         ASSERT(IAM_WRITER_ILL(ill));

1386         if (ill->ill_dlpi_capab_state != IDCS_OK)
1387                 return;

1389         ill->ill_dlpi_capab_state = reneg ? IDCS_RENEG : IDCS_RESET_SENT;

1391         ill_capability_send(ill, ill->ill_capab_reset_mp);
1392         ill->ill_capab_reset_mp = NULL;
1393         /*
1394          * We turn off all capabilities except those pertaining to
1395          * direct function call capabilities viz. ILL_CAPAB_DLD*
1396          * which will be turned off by the corresponding reset functions.
1397          */
1398         ill->ill_capabilities &= ~(ILL_CAPAB_HCKSUM  | ILL_CAPAB_ZEROCOPY);
1399 }

1401 static void
1402 ill_capability_reset_alloc(ill_t *ill)
1403 {
1404         mblk_t *mp;
1405         size_t  size = 0;
1406         int     err;
1407         dl_capability_req_t     *capb;

1409         ASSERT(IAM_WRITER_ILL(ill));
1410         ASSERT(ill->ill_capab_reset_mp == NULL);

1412         if (ILL_HCKSUM_CAPABLE(ill)) {
1413                 size += sizeof (dl_capability_sub_t) +
1414                     sizeof (dl_capab_hcksum_t);
1415         }

1417         if (ill->ill_capabilities & ILL_CAPAB_ZEROCOPY) {
1418                 size += sizeof (dl_capability_sub_t) +
1419                     sizeof (dl_capab_zerocopy_t);
1420         }

1422         if (ill->ill_capabilities & ILL_CAPAB_DLD) {
1423                 size += sizeof (dl_capability_sub_t) +
1424                     sizeof (dl_capab_dld_t);
1425         }

1427         mp = allocb_wait(size + sizeof (dl_capability_req_t), BPRI_MED,
1428             STR_NOSIG, &err);

1430         mp->b_datap->db_type = M_PROTO;
1431         bzero(mp->b_rptr, size + sizeof (dl_capability_req_t));

1433         capb = (dl_capability_req_t *)mp->b_rptr;
1434         capb->dl_primitive = DL_CAPABILITY_REQ;
1435         capb->dl_sub_offset = sizeof (dl_capability_req_t);
1436         capb->dl_sub_length = size;

1438         mp->b_wptr += sizeof (dl_capability_req_t);

1440         /*
1441          * Each handler fills in the corresponding dl_capability_sub_t
1442          * inside the mblk,
1443          */
1444         ill_capability_hcksum_reset_fill(ill, mp);
1445         ill_capability_zerocopy_reset_fill(ill, mp);
1446         ill_capability_dld_reset_fill(ill, mp);
```

```
1448         ill->ill_capab_reset_mp = mp;
1449 }

1451 static void
1452 ill_capability_id_ack(ill_t *ill, mblk_t *mp, dl_capability_sub_t *outers)
1453 {
1454         dl_capab_id_t *id_ic;
1455         uint_t sub_dl_cap = outers->dl_cap;
1456         dl_capability_sub_t *inners;
1457         uint8_t *capend;

1459         ASSERT(sub_dl_cap == DL_CAPAB_ID_WRAPPER);

1461         /*
1462          * Note: range checks here are not absolutely sufficient to
1463          * make us robust against malformed messages sent by drivers;
1464          * this is in keeping with the rest of IP's dlpi handling.
1465          * (Remember, it's coming from something else in the kernel
1466          * address space)
1467          */

1469         capend = (uint8_t *)(outers + 1) + outers->dl_length;
1470         if (capend > mp->b_wptr) {
1471                 cmn_err(CE_WARN, "ill_capability_id_ack: "
1472                     "malformed sub-capability too long for mblk");
1473                 return;
1474         }

1476         id_ic = (dl_capab_id_t *)(outers + 1);

1478         if (outers->dl_length < sizeof (*id_ic) ||
1479             (inners = &id_ic->id_subcap,
1480             inners->dl_length > (outers->dl_length - sizeof (*inners)))) {
1481                 cmn_err(CE_WARN, "ill_capability_id_ack: malformed "
1482                     "encapsulated capab type %d too long for mblk",
1483                     inners->dl_cap);
1484                 return;
1485         }

1487         if (!dlcapabcheckqid(&id_ic->id_mid, ill->ill_lmod_rq)) {
1488                 ip1dbg(("ill_capability_id_ack: mid token for capab type %d "
1489                     "isn't as expected; pass-thru module(s) detected, "
1490                     "discarding capability\n", inners->dl_cap));
1491                 return;
1492         }

1494         /* Process the encapsulated sub-capability */
1495         ill_capability_dispatch(ill, mp, inners);
1496 }

1498 static void
1499 ill_capability_dld_reset_fill(ill_t *ill, mblk_t *mp)
1500 {
1501         dl_capability_sub_t *dl_subcap;

1503         if (!(ill->ill_capabilities & ILL_CAPAB_DLD))
1504                 return;

1506         /*
1507          * The dl_capab_dld_t that follows the dl_capability_sub_t is not
1508          * initialized below since it is not used by DLD.
1509          */
1510         dl_subcap = (dl_capability_sub_t *)mp->b_wptr;
1511         dl_subcap->dl_cap = DL_CAPAB_DLD;
1512         dl_subcap->dl_length = sizeof (dl_capab_dld_t);
```



```
1514         mp->b_wptr += sizeof (dl_capability_sub_t) + sizeof (dl_capab_dld_t);
1515 }

1517 static void
1518 ill_capability_dispatch(ill_t *ill, mblk_t *mp, dl_capability_sub_t *subp)
1519 {
1520         /*
1521          * If no ipif was brought up over this ill, this DL_CAPABILITY_REQ/ACK
1522          * is only to get the VRRP capability.
1523          *
1524          * Note that we cannot check ill_ipif_up_count here since
1525          * ill_ipif_up_count is only incremented when the resolver is setup.
1526          * That is done asynchronously, and can race with this function.
1527          */
1528         if (!ill->ill_dl_up) {
1529                 if (subp->dl_cap == DL_CAPAB_VRRP)
1530                         ill_capability_vrrp_ack(ill, mp, subp);
1531                 return;
1532         }

1534         switch (subp->dl_cap) {
1535         case DL_CAPAB_HCKSUM:
1536                 ill_capability_hcksum_ack(ill, mp, subp);
1537                 break;
1538         case DL_CAPAB_ZEROCOPY:
1539                 ill_capability_zerocopy_ack(ill, mp, subp);
1540                 break;
1541         case DL_CAPAB_DLD:
1542                 ill_capability_dld_ack(ill, mp, subp);
1543                 break;
1544         case DL_CAPAB_VRRP:
1545                 break;
1546         default:
1547                 ip1dbg(("ill_capability_dispatch: unknown capab type %d\n",
1548                     subp->dl_cap));
1549         }
1550 }

1552 /*
1553  * Process the vrrp capability received from a DLS Provider. isub must point
1554  * to the sub-capability (DL_CAPAB_VRRP) of a DL_CAPABILITY_ACK message.
1555  */
1556 static void
1557 ill_capability_vrrp_ack(ill_t *ill, mblk_t *mp, dl_capability_sub_t *isub)
1558 {
1559         dl_capab_vrrp_t *vrrp;
1560         uint_t          sub_dl_cap = isub->dl_cap;
1561         uint8_t         *capend;

1563         ASSERT(IAM_WRITER_ILL(ill));
1564         ASSERT(sub_dl_cap == DL_CAPAB_VRRP);

1566         /*
1567          * Note: range checks here are not absolutely sufficient to
1568          * make us robust against malformed messages sent by drivers;
1569          * this is in keeping with the rest of IP's dlpi handling.
1570          * (Remember, it's coming from something else in the kernel
1571          * address space)
1572          */
1573         capend = (uint8_t *)(isub + 1) + isub->dl_length;
1574         if (capend > mp->b_wptr) {
1575                 cmn_err(CE_WARN, "ill_capability_vrrp_ack: "
1576                     "malformed sub-capability too long for mblk");
1577                 return;
1578         }
1579         vrrp = (dl_capab_vrrp_t *)(isub + 1);
```

```
1581         /*
1582          * Compare the IP address family and set ILLF_VRRP for the right ill.
1583          */
1584         if ((vrrp->vrrp_af == AF_INET6 && ill->ill_isv6) ||
1585             (vrrp->vrrp_af == AF_INET && !ill->ill_isv6)) {
1586                 ill->ill_flags |= ILLF_VRRP;
1587         }
1588 }

1590 /*
1591  * Process a hardware checksum offload capability negotiation ack received
1592  * from a DLS Provider.isub must point to the sub-capability (DL_CAPAB_HCKSUM)
1593  * of a DL_CAPABILITY_ACK message.
1594  */
1595 static void
1596 ill_capability_hcksum_ack(ill_t *ill, mblk_t *mp, dl_capability_sub_t *isub)
1597 {
1598         dl_capability_req_t     *ocap;
1599         dl_capab_hcksum_t       *ihck, *ohck;
1600         ill_hcksum_capab_t      **ill_hcksum;
1601         mblk_t                  *nmp = NULL;
1602         uint_t                  sub_dl_cap = isub->dl_cap;
1603         uint8_t                 *capend;

1605         ASSERT(sub_dl_cap == DL_CAPAB_HCKSUM);

1607         ill_hcksum = (ill_hcksum_capab_t **)&ill->ill_hcksum_capab;

1609         /*
1610          * Note: range checks here are not absolutely sufficient to
1611          * make us robust against malformed messages sent by drivers;
1612          * this is in keeping with the rest of IP's dlpi handling.
1613          * (Remember, it's coming from something else in the kernel
1614          * address space)
1615          */
1616         capend = (uint8_t *)(isub + 1) + isub->dl_length;
1617         if (capend > mp->b_wptr) {
1618                 cmn_err(CE_WARN, "ill_capability_hcksum_ack: "
1619                     "malformed sub-capability too long for mblk");
1620                 return;
1621         }

1623         /*
1624          * There are two types of acks we process here:
1625          * 1. acks in reply to a (first form) generic capability req
1626          *    (no ENABLE flag set)
1627          * 2. acks in reply to a ENABLE capability req.
1628          *    (ENABLE flag set)
1629          */
1630         ihck = (dl_capab_hcksum_t *)(isub + 1);

1632         if (ihck->hcksum_version != HCKSUM_VERSION_1) {
1633                 cmn_err(CE_CONT, "ill_capability_hcksum_ack: "
1634                     "unsupported hardware checksum "
1635                     "sub-capability (version %d, expected %d)",
1636                     ihck->hcksum_version, HCKSUM_VERSION_1);
1637                 return;
1638         }

1640         if (!dlcapabcheckqid(&ihck->hcksum_mid, ill->ill_lmod_rq)) {
1641                 ip1dbg(("ill_capability_hcksum_ack: mid token for hardware "
1642                     "checksum capability isn't as expected; pass-thru "
1643                     "module(s) detected, discarding capability\n"));
1644                 return;
1645         }
```



```
1647 #define CURR_HCKSUM_CAPAB                                       \
1648         (HCKSUM_INET_PARTIAL | HCKSUM_INET_FULL_V4 |    \
1649         HCKSUM_INET_FULL_V6 | HCKSUM_IPHDRCKSUM)

1651         if ((ihck->hcksum_txflags & HCKSUM_ENABLE) &&
1652             (ihck->hcksum_txflags & CURR_HCKSUM_CAPAB)) {
1653                 /* do ENABLE processing */
1654                 if (*ill_hcksum == NULL) {
1655                         *ill_hcksum = kmem_zalloc(sizeof (ill_hcksum_capab_t),
1656                             KM_NOSLEEP);

1658                         if (*ill_hcksum == NULL) {
1659                                 cmn_err(CE_WARN, "ill_capability_hcksum_ack: "
1660                                     "could not enable hcksum version %d "
1661                                     "for %s (ENOMEM)\n", HCKSUM_CURRENT_VERSION,
1662                                     ill->ill_name);
1663                                 return;
1664                         }
1665                 }

1667                 (*ill_hcksum)->ill_hcksum_version = ihck->hcksum_version;
1668                 (*ill_hcksum)->ill_hcksum_txflags = ihck->hcksum_txflags;
1669                 ill->ill_capabilities |= ILL_CAPAB_HCKSUM;
1670                 ip1dbg(("ill_capability_hcksum_ack: interface %s "
1671                     "has enabled hardware checksumming\n ",
1672                     ill->ill_name));
1673         } else if (ihck->hcksum_txflags & CURR_HCKSUM_CAPAB) {
1674                 /*
1675                  * Enabling hardware checksum offload
1676                  * Currently IP supports {TCP,UDP}/IPv4
1677                  * partial and full cksum offload and
1678                  * IPv4 header checksum offload.
1679                  * Allocate new mblk which will
1680                  * contain a new capability request
1681                  * to enable hardware checksum offload.
1682                  */
1683                 uint_t  size;
1684                 uchar_t *rptr;

1686                 size = sizeof (dl_capability_req_t) +
1687                     sizeof (dl_capability_sub_t) + isub->dl_length;

1689                 if ((nmp = ip_dlpi_alloc(size, DL_CAPABILITY_REQ)) == NULL) {
1690                         cmn_err(CE_WARN, "ill_capability_hcksum_ack: "
1691                             "could not enable hardware cksum for %s (ENOMEM)\n",
1692                             ill->ill_name);
1693                         return;
1694                 }

1696                 rptr = nmp->b_rptr;
1697                 /* initialize dl_capability_req_t */
1698                 ocap = (dl_capability_req_t *)nmp->b_rptr;
1699                 ocap->dl_sub_offset =
1700                     sizeof (dl_capability_req_t);
1701                 ocap->dl_sub_length =
1702                     sizeof (dl_capability_sub_t) +
1703                     isub->dl_length;
1704                 nmp->b_rptr += sizeof (dl_capability_req_t);

1706                 /* initialize dl_capability_sub_t */
1707                 bcopy(isub, nmp->b_rptr, sizeof (*isub));
1708                 nmp->b_rptr += sizeof (*isub);

1710                 /* initialize dl_capab_hcksum_t */
1711                 ohck = (dl_capab_hcksum_t *)nmp->b_rptr;
```

```
1712                bcopy(ihck, ohck, sizeof (*ihck));

1714                nmp->b_rptr = rptr;
1715                ASSERT(nmp->b_wptr == (nmp->b_rptr + size));

1717                /* Set ENABLE flag */
1718                ohck->hcksum_txflags &= CURR_HCKSUM_CAPAB;
1719                ohck->hcksum_txflags |= HCKSUM_ENABLE;

1721                /*
1722                 * nmp points to a DL_CAPABILITY_REQ message to enable
1723                 * hardware checksum acceleration.
1724                 */
1725                ill_capability_send(ill, nmp);
1726        } else {
1727                ip1dbg(("ill_capability_hcksum_ack: interface %s has "
1728                    "advertised %x hardware checksum capability flags\n",
1729                    ill->ill_name, ihck->hcksum_txflags));
1730        }
1731 }

1733 static void
1734 ill_capability_hcksum_reset_fill(ill_t *ill, mblk_t *mp)
1735 {
1736        dl_capab_hcksum_t *hck_subcap;
1737        dl_capability_sub_t *dl_subcap;

1739        if (!ILL_HCKSUM_CAPABLE(ill))
1740                return;

1742        ASSERT(ill->ill_hcksum_capab != NULL);

1744        dl_subcap = (dl_capability_sub_t *)mp->b_wptr;
1745        dl_subcap->dl_cap = DL_CAPAB_HCKSUM;
1746        dl_subcap->dl_length = sizeof (*hck_subcap);

1748        hck_subcap = (dl_capab_hcksum_t *)(dl_subcap + 1);
1749        hck_subcap->hcksum_version = ill->ill_hcksum_capab->ill_hcksum_version;
1750        hck_subcap->hcksum_txflags = 0;

1752        mp->b_wptr += sizeof (*dl_subcap) + sizeof (*hck_subcap);
1753 }

1755 static void
1756 ill_capability_zerocopy_ack(ill_t *ill, mblk_t *mp, dl_capability_sub_t *isub)
1757 {
1758        mblk_t *nmp = NULL;
1759        dl_capability_req_t *oc;
1760        dl_capab_zerocopy_t *zc_ic, *zc_oc;
1761        ill_zerocopy_capab_t **ill_zerocopy_capab;
1762        uint_t sub_dl_cap = isub->dl_cap;
1763        uint8_t *capend;

1765        ASSERT(sub_dl_cap == DL_CAPAB_ZEROCOPY);

1767        ill_zerocopy_capab = (ill_zerocopy_capab_t **)&ill->ill_zerocopy_capab;

1769        /*
1770         * Note: range checks here are not absolutely sufficient to
1771         * make us robust against malformed messages sent by drivers;
1772         * this is in keeping with the rest of IP's dlpi handling.
1773         * (Remember, it's coming from something else in the kernel
1774         * address space)
1775         */
1776        capend = (uint8_t *)(isub + 1) + isub->dl_length;
1777        if (capend > mp->b_wptr) {
```



```
1778                cmn_err(CE_WARN, "ill_capability_zerocopy_ack: "
1779                    "malformed sub-capability too long for mblk");
1780                return;
1781        }

1783        zc_ic = (dl_capab_zerocopy_t *)(isub + 1);
1784        if (zc_ic->zerocopy_version != ZEROCOPY_VERSION_1) {
1785                cmn_err(CE_CONT, "ill_capability_zerocopy_ack: "
1786                    "unsupported ZEROCOPY sub-capability (version %d, "
1787                    "expected %d)", zc_ic->zerocopy_version,
1788                    ZEROCOPY_VERSION_1);
1789                return;
1790        }

1792        if (!dlcapabcheckqid(&zc_ic->zerocopy_mid, ill->ill_lmod_rq)) {
1793                ip1dbg(("ill_capability_zerocopy_ack: mid token for zerocopy "
1794                    "capability isn't as expected; pass-thru module(s) "
1795                    "detected, discarding capability\n"));
1796                return;
1797        }

1799        if ((zc_ic->zerocopy_flags & DL_CAPAB_VMSAFE_MEM) != 0) {
1800                if (*ill_zerocopy_capab == NULL) {
1801                        *ill_zerocopy_capab =
1802                            kmem_zalloc(sizeof (ill_zerocopy_capab_t),
1803                            KM_NOSLEEP);

1805                        if (*ill_zerocopy_capab == NULL) {
1806                                cmn_err(CE_WARN, "ill_capability_zerocopy_ack: "
1807                                    "could not enable Zero-copy version %d "
1808                                    "for %s (ENOMEM)\n", ZEROCOPY_VERSION_1,
1809                                    ill->ill_name);
1810                                return;
1811                        }
1812                }

1814                ip1dbg(("ill_capability_zerocopy_ack: interface %s "
1815                    "supports Zero-copy version %d\n", ill->ill_name,
1816                    ZEROCOPY_VERSION_1));

1818                (*ill_zerocopy_capab)->ill_zerocopy_version =
1819                    zc_ic->zerocopy_version;
1820                (*ill_zerocopy_capab)->ill_zerocopy_flags =
1821                    zc_ic->zerocopy_flags;

1823                ill->ill_capabilities |= ILL_CAPAB_ZEROCOPY;
1824        } else {
1825                uint_t size;
1826                uchar_t *rptr;

1828                size = sizeof (dl_capability_req_t) +
1829                    sizeof (dl_capability_sub_t) +
1830                    sizeof (dl_capab_zerocopy_t);

1832                if ((nmp = ip_dlpi_alloc(size, DL_CAPABILITY_REQ)) == NULL) {
1833                        cmn_err(CE_WARN, "ill_capability_zerocopy_ack: "
1834                            "could not enable zerocopy for %s (ENOMEM)\n",
1835                            ill->ill_name);
1836                        return;
1837                }

1839                rptr = nmp->b_rptr;
1840                /* initialize dl_capability_req_t */
1841                oc = (dl_capability_req_t *)rptr;
1842                oc->dl_sub_offset = sizeof (dl_capability_req_t);
1843                oc->dl_sub_length = sizeof (dl_capability_sub_t) +
```

```
1844                    sizeof (dl_capab_zerocopy_t);
1845                rptr += sizeof (dl_capability_req_t);

1847                /* initialize dl_capability_sub_t */
1848                bcopy(isub, rptr, sizeof (*isub));
1849                rptr += sizeof (*isub);

1851                /* initialize dl_capab_zerocopy_t */
1852                zc_oc = (dl_capab_zerocopy_t *)rptr;
1853                *zc_oc = *zc_ic;

1855                ip1dbg(("ill_capability_zerocopy_ack: asking interface %s "
1856                    "to enable zero-copy version %d\n", ill->ill_name,
1857                    ZEROCOPY_VERSION_1));

1859                /* set VMSAFE_MEM flag */
1860                zc_oc->zerocopy_flags |= DL_CAPAB_VMSAFE_MEM;

1862                /* nmp points to a DL_CAPABILITY_REQ message to enable zcopy */
1863                ill_capability_send(ill, nmp);
1864        }
1865 }

1867 static void
1868 ill_capability_zerocopy_reset_fill(ill_t *ill, mblk_t *mp)
1869 {
1870        dl_capab_zerocopy_t *zerocopy_subcap;
1871        dl_capability_sub_t *dl_subcap;

1873        if (!(ill->ill_capabilities & ILL_CAPAB_ZEROCOPY))
1874                return;

1876        ASSERT(ill->ill_zerocopy_capab != NULL);

1878        dl_subcap = (dl_capability_sub_t *)mp->b_wptr;
1879        dl_subcap->dl_cap = DL_CAPAB_ZEROCOPY;
1880        dl_subcap->dl_length = sizeof (*zerocopy_subcap);

1882        zerocopy_subcap = (dl_capab_zerocopy_t *)(dl_subcap + 1);
1883        zerocopy_subcap->zerocopy_version =
1884            ill->ill_zerocopy_capab->ill_zerocopy_version;
1885        zerocopy_subcap->zerocopy_flags = 0;

1887        mp->b_wptr += sizeof (*dl_subcap) + sizeof (*zerocopy_subcap);
1888 }

1890 /*
1891  * DLD capability
1892  * Refer to dld.h for more information regarding the purpose and usage
1893  * of this capability.
1894  */
1895 static void
1896 ill_capability_dld_ack(ill_t *ill, mblk_t *mp, dl_capability_sub_t *isub)
1897 {
1898        dl_capab_dld_t          *dld_ic, dld;
1899        uint_t                  sub_dl_cap = isub->dl_cap;
1900        uint8_t                 *capend;
1901        ill_dld_capab_t         *idc;

1903        ASSERT(IAM_WRITER_ILL(ill));
1904        ASSERT(sub_dl_cap == DL_CAPAB_DLD);

1906        /*
1907         * Note: range checks here are not absolutely sufficient to
1908         * make us robust against malformed messages sent by drivers;
1909         * this is in keeping with the rest of IP's dlpi handling.
```



```
1910         * (Remember, it's coming from something else in the kernel
1911         * address space)
1912         */
1913        capend = (uint8_t *)(isub + 1) + isub->dl_length;
1914        if (capend > mp->b_wptr) {
1915                cmn_err(CE_WARN, "ill_capability_dld_ack: "
1916                    "malformed sub-capability too long for mblk");
1917                return;
1918        }
1919        dld_ic = (dl_capab_dld_t *)(isub + 1);
1920        if (dld_ic->dld_version != DLD_CURRENT_VERSION) {
1921                cmn_err(CE_CONT, "ill_capability_dld_ack: "
1922                    "unsupported DLD sub-capability (version %d, "
1923                    "expected %d)", dld_ic->dld_version,
1924                    DLD_CURRENT_VERSION);
1925                return;
1926        }
1927        if (!dlcapabcheckqid(&dld_ic->dld_mid, ill->ill_lmod_rq)) {
1928                ip1dbg(("ill_capability_dld_ack: mid token for dld "
1929                    "capability isn't as expected; pass-thru module(s) "
1930                    "detected, discarding capability\n"));
1931                return;
1932        }

1934        /*
1935         * Copy locally to ensure alignment.
1936         */
1937        bcopy(dld_ic, &dld, sizeof (dl_capab_dld_t));

1939        if ((idc = ill->ill_dld_capab) == NULL) {
1940                idc = kmem_zalloc(sizeof (ill_dld_capab_t), KM_NOSLEEP);
1941                if (idc == NULL) {
1942                        cmn_err(CE_WARN, "ill_capability_dld_ack: "
1943                            "could not enable DLD version %d "
1944                            "for %s (ENOMEM)\n", DLD_CURRENT_VERSION,
1945                            ill->ill_name);
1946                        return;
1947                }
1948                ill->ill_dld_capab = idc;
1949        }
1950        idc->idc_capab_df = (ip_capab_func_t)dld.dld_capab;
1951        idc->idc_capab_dh = (void *)dld.dld_capab_handle;
1952        ip1dbg(("ill_capability_dld_ack: interface %s "
1953            "supports DLD version %d\n", ill->ill_name, DLD_CURRENT_VERSION));

1955        ill_capability_dld_enable(ill);
1956 }

1958 /*
1959  * Typically capability negotiation between IP and the driver happens via
1960  * DLPI message exchange. However GLD also offers a direct function call
1961  * mechanism to exchange the DLD_DIRECT_CAPAB and DLD_POLL_CAPAB capabilities,
1962  * But arbitrary function calls into IP or GLD are not permitted, since both
1963  * of them are protected by their own perimeter mechanism. The perimeter can
1964  * be viewed as a coarse lock or serialization mechanism. The hierarchy of
1965  * these perimeters is IP -> MAC. Thus for example to enable the squeue
1966  * polling, IP needs to enter its perimeter, then call ill_mac_perim_enter
1967  * to enter the mac perimeter and then do the direct function calls into
1968  * GLD to enable squeue polling. The ring related callbacks from the mac into
1969  * the stack to add, bind, quiesce, restart or cleanup a ring are all
1970  * protected by the mac perimeter.
1971  */
1972 static void
1973 ill_mac_perim_enter(ill_t *ill, mac_perim_handle_t *mphp)
1974 {
1975        ill_dld_capab_t         *idc = ill->ill_dld_capab;
```

```
1976         int                     err;

1978         err = idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_PERIM, mphp,
1979             DLD_ENABLE);
1980         ASSERT(err == 0);
1981 }

1983 static void
1984 ill_mac_perim_exit(ill_t *ill, mac_perim_handle_t mph)
1985 {
1986         ill_dld_capab_t         *idc = ill->ill_dld_capab;
1987         int                     err;

1989         err = idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_PERIM, mph,
1990             DLD_DISABLE);
1991         ASSERT(err == 0);
1992 }

1994 boolean_t
1995 ill_mac_perim_held(ill_t *ill)
1996 {
1997         ill_dld_capab_t         *idc = ill->ill_dld_capab;

1999         return (idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_PERIM, NULL,
2000             DLD_QUERY));
2001 }

2003 static void
2004 ill_capability_direct_enable(ill_t *ill)
2005 {
2006         ill_dld_capab_t         *idc = ill->ill_dld_capab;
2007         ill_dld_direct_t        *idd = &idc->idc_direct;
2008         dld_capab_direct_t      direct;
2009         int                     rc;

2011         ASSERT(!ill->ill_isv6 && IAM_WRITER_ILL(ill));

2013         bzero(&direct, sizeof (direct));
2014         direct.di_rx_cf = (uintptr_t)ip_input;
2015         direct.di_rx_ch = ill;

2017         rc = idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_DIRECT, &direct,
2018             DLD_ENABLE);
2019         if (rc == 0) {
2020                 idd->idd_tx_df = (ip_dld_tx_t)direct.di_tx_df;
2021                 idd->idd_tx_dh = direct.di_tx_dh;
2022                 idd->idd_tx_cb_df = (ip_dld_callb_t)direct.di_tx_cb_df;
2023                 idd->idd_tx_cb_dh = direct.di_tx_cb_dh;
2024                 idd->idd_tx_fctl_df = (ip_dld_fctl_t)direct.di_tx_fctl_df;
2025                 idd->idd_tx_fctl_dh = direct.di_tx_fctl_dh;
2026                 ASSERT(idd->idd_tx_cb_df != NULL);
2027                 ASSERT(idd->idd_tx_fctl_df != NULL);
2028                 ASSERT(idd->idd_tx_df != NULL);
2029                 /*
2030                  * One time registration of flow enable callback function
2031                  */
2032                 ill->ill_flownotify_mh = idd->idd_tx_cb_df(idd->idd_tx_cb_dh,
2033                     ill_flow_enable, ill);
2034                 ill->ill_capabilities |= ILL_CAPAB_DLD_DIRECT;
2035                 DTRACE_PROBE1(direct_on, (ill_t *), ill);
2036         } else {
2037                 cmn_err(CE_WARN, "warning: could not enable DIRECT "
2038                     "capability, rc = %d\n", rc);
2039                 DTRACE_PROBE2(direct_off, (ill_t *), ill, (int), rc);
2040         }
2041 }
```



```
2043 static void
2044 ill_capability_poll_enable(ill_t *ill)
2045 {
2046         ill_dld_capab_t         *idc = ill->ill_dld_capab;
2047         dld_capab_poll_t        poll;
2048         int                     rc;

2050         ASSERT(!ill->ill_isv6 && IAM_WRITER_ILL(ill));

2052         bzero(&poll, sizeof (poll));
2053         poll.poll_ring_add_cf = (uintptr_t)ip_squeue_add_ring;
2054         poll.poll_ring_remove_cf = (uintptr_t)ip_squeue_clean_ring;
2055         poll.poll_ring_quiesce_cf = (uintptr_t)ip_squeue_quiesce_ring;
2056         poll.poll_ring_restart_cf = (uintptr_t)ip_squeue_restart_ring;
2057         poll.poll_ring_bind_cf = (uintptr_t)ip_squeue_bind_ring;
2058         poll.poll_ring_ch = ill;
2059         rc = idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_POLL, &poll,
2060             DLD_ENABLE);
2061         if (rc == 0) {
2062                 ill->ill_capabilities |= ILL_CAPAB_DLD_POLL;
2063                 DTRACE_PROBE1(poll_on, (ill_t *), ill);
2064         } else {
2065                 ip1dbg(("warning: could not enable POLL "
2066                     "capability, rc = %d\n", rc));
2067                 DTRACE_PROBE2(poll_off, (ill_t *), ill, (int), rc);
2068         }
2069 }

2071 /*
2072  * Enable the LSO capability.
2073  */
2074 static void
2075 ill_capability_lso_enable(ill_t *ill)
2076 {
2077         ill_dld_capab_t *idc = ill->ill_dld_capab;
2078         dld_capab_lso_t lso;
2079         int rc;

2081         ASSERT(!ill->ill_isv6 && IAM_WRITER_ILL(ill));

2083         if (ill->ill_lso_capab == NULL) {
2084                 ill->ill_lso_capab = kmem_zalloc(sizeof (ill_lso_capab_t),
2085                     KM_NOSLEEP);
2086                 if (ill->ill_lso_capab == NULL) {
2087                         cmn_err(CE_WARN, "ill_capability_lso_enable: "
2088                             "could not enable LSO for %s (ENOMEM)\n",
2089                             ill->ill_name);
2090                         return;
2091                 }
2092         }

2094         bzero(&lso, sizeof (lso));
2095         if ((rc = idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_LSO, &lso,
2096             DLD_ENABLE)) == 0) {
2097                 ill->ill_lso_capab->ill_lso_flags = lso.lso_flags;
2098                 ill->ill_lso_capab->ill_lso_max = lso.lso_max;
2099                 ill->ill_capabilities |= ILL_CAPAB_LSO;
2100                 ip1dbg(("ill_capability_lso_enable: interface %s "
2101                     "has enabled LSO\n ", ill->ill_name));
2102         } else {
2103                 kmem_free(ill->ill_lso_capab, sizeof (ill_lso_capab_t));
2104                 ill->ill_lso_capab = NULL;
2105                 DTRACE_PROBE2(lso_off, (ill_t *), ill, (int), rc);
2106         }
2107 }
```

```
2109 static void
2110 ill_capability_dld_enable(ill_t *ill)
2111 {
2112         mac_perim_handle_t mph;

2114         ASSERT(IAM_WRITER_ILL(ill));

2116         if (ill->ill_isv6)
2117                 return;

2119         ill_mac_perim_enter(ill, &mph);
2120         if (!ill->ill_isv6) {
2121                 ill_capability_direct_enable(ill);
2122                 ill_capability_poll_enable(ill);
2123                 ill_capability_lso_enable(ill);
2124         }
2125         ill->ill_capabilities |= ILL_CAPAB_DLD;
2126         ill_mac_perim_exit(ill, mph);
2127 }

2129 static void
2130 ill_capability_dld_disable(ill_t *ill)
2131 {
2132         ill_dld_capab_t *idc;
2133         ill_dld_direct_t *idd;
2134         mac_perim_handle_t      mph;

2136         ASSERT(IAM_WRITER_ILL(ill));

2138         if (!(ill->ill_capabilities & ILL_CAPAB_DLD))
2139                 return;

2141         ill_mac_perim_enter(ill, &mph);

2143         idc = ill->ill_dld_capab;
2144         if ((ill->ill_capabilities & ILL_CAPAB_DLD_DIRECT) != 0) {
2145                 /*
2146                  * For performance we avoid locks in the transmit data path
2147                  * and don't maintain a count of the number of threads using
2148                  * direct calls. Thus some threads could be using direct
2149                  * transmit calls to GLD, even after the capability mechanism
2150                  * turns it off. This is still safe since the handles used in
2151                  * the direct calls continue to be valid until the unplumb is
2152                  * completed. Remove the callback that was added (1-time) at
2153                  * capab enable time.
2154                  */
2155                 mutex_enter(&ill->ill_lock);
2156                 ill->ill_capabilities &= ~ILL_CAPAB_DLD_DIRECT;
2157                 mutex_exit(&ill->ill_lock);
2158                 if (ill->ill_flownotify_mh != NULL) {
2159                         idd = &idc->idc_direct;
2160                         idd->idd_tx_cb_df(idd->idd_tx_cb_dh, NULL,
2161                             ill->ill_flownotify_mh);
2162                         ill->ill_flownotify_mh = NULL;
2163                 }
2164                 (void) idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_DIRECT,
2165                     NULL, DLD_DISABLE);
2166         }

2168         if ((ill->ill_capabilities & ILL_CAPAB_DLD_POLL) != 0) {
2169                 ill->ill_capabilities &= ~ILL_CAPAB_DLD_POLL;
2170                 ip_squeue_clean_all(ill);
2171                 (void) idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_POLL,
2172                     NULL, DLD_DISABLE);
2173         }
```



```
2175         if ((ill->ill_capabilities & ILL_CAPAB_LSO) != 0) {
2176                 ASSERT(ill->ill_lso_capab != NULL);
2177                 /*
2178                  * Clear the capability flag for LSO but retain the
2179                  * ill_lso_capab structure since it's possible that another
2180                  * thread is still referring to it.  The structure only gets
2181                  * deallocated when we destroy the ill.
2182                  */

2184                 ill->ill_capabilities &= ~ILL_CAPAB_LSO;
2185                 (void) idc->idc_capab_df(idc->idc_capab_dh, DLD_CAPAB_LSO,
2186                     NULL, DLD_DISABLE);
2187         }

2189         ill->ill_capabilities &= ~ILL_CAPAB_DLD;
2190         ill_mac_perim_exit(ill, mph);
2191 }

2193 /*
2194  * Capability Negotiation protocol
2195  *
2196  * We don't wait for DLPI capability operations to finish during interface
2197  * bringup or teardown. Doing so would introduce more asynchrony and the
2198  * interface up/down operations will need multiple return and restarts.
2199  * Instead the 'ipsq_current_ipif' of the ipsq is not cleared as long as
2200  * the 'ill_dlpi_deferred' chain is non-empty. This ensures that the next
2201  * exclusive operation won't start until the DLPI operations of the previous
2202  * exclusive operation complete.
2203  *
2204  * The capability state machine is shown below.
2205  *
2206  * state                next state              event, action
2207  *
2208  * IDCS_UNKNOWN         IDCS_PROBE_SENT         ill_capability_probe
2209  * IDCS_PROBE_SENT      IDCS_OK                 ill_capability_ack
2210  * IDCS_PROBE_SENT      IDCS_FAILED             ip_rput_dlpi_writer (nack)
2211  * IDCS_OK              IDCS_RENEG              Receipt of DL_NOTE_CAPAB_RENEG
2212  * IDCS_OK              IDCS_RESET_SENT         ill_capability_reset
2213  * IDCS_RESET_SENT      IDCS_UNKNOWN            ill_capability_ack_thr
2214  * IDCS_RENEG           IDCS_PROBE_SENT         ill_capability_ack_thr ->
2215  *                                                  ill_capability_probe.
2216  */

2218 /*
2219  * Dedicated thread started from ip_stack_init that handles capability
2220  * disable. This thread ensures the taskq dispatch does not fail by waiting
2221  * for resources using TQ_SLEEP. The taskq mechanism is used to ensure
2222  * that direct calls to DLD are done in a cv_waitable context.
2223  */
2224 void
2225 ill_taskq_dispatch(ip_stack_t *ipst)
2226 {
2227         callb_cpr_t cprinfo;
2228         char    name[64];
2229         mblk_t  *mp;

2231         (void) snprintf(name, sizeof (name), "ill_taskq_dispatch_%d",
2232             ipst->ips_netstack->netstack_stackid);
2233         CALLB_CPR_INIT(&cprinfo, &ipst->ips_capab_taskq_lock, callb_generic_cpr,
2234             name);
2235         mutex_enter(&ipst->ips_capab_taskq_lock);

2237         for (;;) {
2238                 mp = ipst->ips_capab_taskq_head;
2239                 while (mp != NULL) {
```

```
2240                        ipst->ips_capab_taskq_head = mp->b_next;
2241                        if (ipst->ips_capab_taskq_head == NULL)
2242                                ipst->ips_capab_taskq_tail = NULL;
2243                        mutex_exit(&ipst->ips_capab_taskq_lock);
2244                        mp->b_next = NULL;

2246                        VERIFY(taskq_dispatch(system_taskq,
2247                            ill_capability_ack_thr, mp, TQ_SLEEP) != 0);
2248                        mutex_enter(&ipst->ips_capab_taskq_lock);
2249                        mp = ipst->ips_capab_taskq_head;
2250                }

2252                if (ipst->ips_capab_taskq_quit)
2253                        break;
2254                CALLB_CPR_SAFE_BEGIN(&cprinfo);
2255                cv_wait(&ipst->ips_capab_taskq_cv, &ipst->ips_capab_taskq_lock);
2256                CALLB_CPR_SAFE_END(&cprinfo, &ipst->ips_capab_taskq_lock);
2257        }
2258        VERIFY(ipst->ips_capab_taskq_head == NULL);
2259        VERIFY(ipst->ips_capab_taskq_tail == NULL);
2260        CALLB_CPR_EXIT(&cprinfo);
2261        thread_exit();
2262 }

2264 /*
2265  * Consume a new-style hardware capabilities negotiation ack.
2266  * Called via taskq on receipt of DL_CAPABILITY_ACK.
2267  */
2268 static void
2269 ill_capability_ack_thr(void *arg)
2270 {
2271        mblk_t  *mp = arg;
2272        dl_capability_ack_t *capp;
2273        dl_capability_sub_t *subp, *endp;
2274        ill_t   *ill;
2275        boolean_t reneg;

2277        ill = (ill_t *)mp->b_prev;
2278        mp->b_prev = NULL;

2280        VERIFY(ipsq_enter(ill, B_FALSE, CUR_OP) == B_TRUE);

2282        if (ill->ill_dlpi_capab_state == IDCS_RESET_SENT ||
2283            ill->ill_dlpi_capab_state == IDCS_RENEG) {
2284                /*
2285                 * We have received the ack for our DL_CAPAB reset request.
2286                 * There isnt' anything in the message that needs processing.
2287                 * All message based capabilities have been disabled, now
2288                 * do the function call based capability disable.
2289                 */
2290                reneg = ill->ill_dlpi_capab_state == IDCS_RENEG;
2291                ill_capability_dld_disable(ill);
2292                ill->ill_dlpi_capab_state = IDCS_UNKNOWN;
2293                if (reneg)
2294                        ill_capability_probe(ill);
2295                goto done;
2296        }

2298        if (ill->ill_dlpi_capab_state == IDCS_PROBE_SENT)
2299                ill->ill_dlpi_capab_state = IDCS_OK;

2301        capp = (dl_capability_ack_t *)mp->b_rptr;

2303        if (capp->dl_sub_length == 0) {
2304                /* no new-style capabilities */
2305                goto done;
```



```
2306        }

2308        /* make sure the driver supplied correct dl_sub_length */
2309        if ((sizeof (*capp) + capp->dl_sub_length) > MBLKL(mp)) {
2310                ip0dbg(("ill_capability_ack: bad DL_CAPABILITY_ACK, "
2311                    "invalid dl_sub_length (%d)\n", capp->dl_sub_length));
2312                goto done;
2313        }

2315 #define SC(base, offset) (dl_capability_sub_t *)(((uchar_t *)(base))+(offset))
2316        /*
2317         * There are sub-capabilities. Process the ones we know about.
2318         * Loop until we don't have room for another sub-cap header..
2319         */
2320        for (subp = SC(capp, capp->dl_sub_offset),
2321            endp = SC(subp, capp->dl_sub_length - sizeof (*subp));
2322            subp <= endp;
2323            subp = SC(subp, sizeof (dl_capability_sub_t) + subp->dl_length)) {

2325                switch (subp->dl_cap) {
2326                case DL_CAPAB_ID_WRAPPER:
2327                        ill_capability_id_ack(ill, mp, subp);
2328                        break;
2329                default:
2330                        ill_capability_dispatch(ill, mp, subp);
2331                        break;
2332                }
2333        }
2334 #undef SC
2335 done:
2336        inet_freemsg(mp);
2337        ill_capability_done(ill);
2338        ipsq_exit(ill->ill_phyint->phyint_ipsq);
2339 }

2341 /*
2342  * This needs to be started in a taskq thread to provide a cv_waitable
2343  * context.
2344  */
2345 void
2346 ill_capability_ack(ill_t *ill, mblk_t *mp)
2347 {
2348        ip_stack_t      *ipst = ill->ill_ipst;

2350        mp->b_prev = (mblk_t *)ill;
2351        ASSERT(mp->b_next == NULL);

2353        if (taskq_dispatch(system_taskq, ill_capability_ack_thr, mp,
2354            TQ_NOSLEEP) != 0)
2355                return;

2357        /*
2358         * The taskq dispatch failed. Signal the ill_taskq_dispatch thread
2359         * which will do the dispatch using TQ_SLEEP to guarantee success.
2360         */
2361        mutex_enter(&ipst->ips_capab_taskq_lock);
2362        if (ipst->ips_capab_taskq_head == NULL) {
2363                ASSERT(ipst->ips_capab_taskq_tail == NULL);
2364                ipst->ips_capab_taskq_head = mp;
2365        } else {
2366                ipst->ips_capab_taskq_tail->b_next = mp;
2367        }
2368        ipst->ips_capab_taskq_tail = mp;

2370        cv_signal(&ipst->ips_capab_taskq_cv);
2371        mutex_exit(&ipst->ips_capab_taskq_lock);
```

```
2372 }
2373
2374 /*
2375  * This routine is called to scan the fragmentation reassembly table for
2376  * the specified ILL for any packets that are starting to smell.
2377  * dead_interval is the maximum time in seconds that will be tolerated.  It
2378  * will either be the value specified in ip_g_frag_timeout, or zero if the
2379  * ILL is shutting down and it is time to blow everything off.
2380  *
2381  * It returns the number of seconds (as a time_t) that the next frag timer
2382  * should be scheduled for, 0 meaning that the timer doesn't need to be
2383  * re-started.  Note that the method of calculating next_timeout isn't
2384  * entirely accurate since time will flow between the time we grab
2385  * current_time and the time we schedule the next timeout.  This isn't a
2386  * big problem since this is the timer for sending an ICMP reassembly time
2387  * exceeded messages, and it doesn't have to be exactly accurate.
2388  *
2389  * This function is
2390  * sometimes called as writer, although this is not required.
2391  */
2392 time_t
2393 ill_frag_timeout(ill_t *ill, time_t dead_interval)
2394 {
2395         ipfb_t  *ipfb;
2396         ipfb_t  *endp;
2397         ipf_t   *ipf;
2398         ipf_t   *ipfnext;
2399         mblk_t  *mp;
2400         time_t  current_time = gethrestime_sec();
2401         time_t  next_timeout = 0;
2402         uint32_t        hdr_length;
2403         mblk_t  *send_icmp_head;
2404         mblk_t  *send_icmp_head_v6;
2405         ip_stack_t *ipst = ill->ill_ipst;
2406         ip_recv_attr_t iras;
2407
2408         bzero(&iras, sizeof (iras));
2409         iras.ira_flags = 0;
2410         iras.ira_ill = iras.ira_rill = ill;
2411         iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
2412         iras.ira_rifindex = iras.ira_ruifindex;
2413
2414         ipfb = ill->ill_frag_hash_tbl;
2415         if (ipfb == NULL)
2416                 return (B_FALSE);
2417         endp = &ipfb[ILL_FRAG_HASH_TBL_COUNT];
2418         /* Walk the frag hash table. */
2419         for (; ipfb < endp; ipfb++) {
2420                 send_icmp_head = NULL;
2421                 send_icmp_head_v6 = NULL;
2422                 mutex_enter(&ipfb->ipfb_lock);
2423                 while ((ipf = ipfb->ipfb_ipf) != 0) {
2424                         time_t frag_time = current_time - ipf->ipf_timestamp;
2425                         time_t frag_timeout;
2426
2427                         if (frag_time < dead_interval) {
2428                                 /*
2429                                  * There are some outstanding fragments
2430                                  * that will timeout later.  Make note of
2431                                  * the time so that we can reschedule the
2432                                  * next timeout appropriately.
2433                                  */
2434                                 frag_timeout = dead_interval - frag_time;
2435                                 if (next_timeout == 0 ||
2436                                     frag_timeout < next_timeout) {
2437                                         next_timeout = frag_timeout;
```



```
2438                                 }
2439                                 break;
2440                         }
2441                         /* Time's up.  Get it out of here. */
2442                         hdr_length = ipf->ipf_nf_hdr_len;
2443                         ipfnext = ipf->ipf_hash_next;
2444                         if (ipfnext)
2445                                 ipfnext->ipf_ptphn = ipf->ipf_ptphn;
2446                         *ipf->ipf_ptphn = ipfnext;
2447                         mp = ipf->ipf_mp->b_cont;
2448                         for (; mp; mp = mp->b_cont) {
2449                                 /* Extra points for neatness. */
2450                                 IP_REASS_SET_START(mp, 0);
2451                                 IP_REASS_SET_END(mp, 0);
2452                         }
2453                         mp = ipf->ipf_mp->b_cont;
2454                         atomic_add_32(&ill->ill_frag_count, -ipf->ipf_count);
2455                         ASSERT(ipfb->ipfb_count >= ipf->ipf_count);
2456                         ipfb->ipfb_count -= ipf->ipf_count;
2457                         ASSERT(ipfb->ipfb_frag_pkts > 0);
2458                         ipfb->ipfb_frag_pkts--;
2459                         /*
2460                          * We do not send any icmp message from here because
2461                          * we currently are holding the ipfb_lock for this
2462                          * hash chain. If we try and send any icmp messages
2463                          * from here we may end up via a put back into ip
2464                          * trying to get the same lock, causing a recursive
2465                          * mutex panic. Instead we build a list and send all
2466                          * the icmp messages after we have dropped the lock.
2467                          */
2468                         if (ill->ill_isv6) {
2469                                 if (hdr_length != 0) {
2470                                         mp->b_next = send_icmp_head_v6;
2471                                         send_icmp_head_v6 = mp;
2472                                 } else {
2473                                         freemsg(mp);
2474                                 }
2475                         } else {
2476                                 if (hdr_length != 0) {
2477                                         mp->b_next = send_icmp_head;
2478                                         send_icmp_head = mp;
2479                                 } else {
2480                                         freemsg(mp);
2481                                 }
2482                         }
2483                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsReasmFails);
2484                         ip_drop_input("ipIfStatsReasmFails", ipf->ipf_mp, ill);
2485                         freeb(ipf->ipf_mp);
2486                 }
2487                 mutex_exit(&ipfb->ipfb_lock);
2488                 /*
2489                  * Now need to send any icmp messages that we delayed from
2490                  * above.
2491                  */
2492                 while (send_icmp_head_v6 != NULL) {
2493                         ip6_t *ip6h;
2494
2495                         mp = send_icmp_head_v6;
2496                         send_icmp_head_v6 = send_icmp_head_v6->b_next;
2497                         mp->b_next = NULL;
2498                         ip6h = (ip6_t *)mp->b_rptr;
2499                         iras.ira_flags = 0;
2500                         /*
2501                          * This will result in an incorrect ALL_ZONES zoneid
2502                          * for multicast packets, but we
2503                          * don't send ICMP errors for those in any case.
```

```
2504                         */
2505                        iras.ira_zoneid =
2506                            ipif_lookup_addr_zoneid_v6(&ip6h->ip6_dst,
2507                            ill, ipst);
2508                        ip_drop_input("ICMP_TIME_EXCEEDED reass", mp, ill);
2509                        icmp_time_exceeded_v6(mp,
2510                            ICMP_REASSEMBLY_TIME_EXCEEDED, B_FALSE,
2511                            &iras);
2512                        ASSERT(!(iras.ira_flags & IRAF_IPSEC_SECURE));
2513                }
2514                while (send_icmp_head != NULL) {
2515                        ipaddr_t dst;

2517                        mp = send_icmp_head;
2518                        send_icmp_head = send_icmp_head->b_next;
2519                        mp->b_next = NULL;

2521                        dst = ((ipha_t *)mp->b_rptr)->ipha_dst;

2523                        iras.ira_flags = IRAF_IS_IPV4;
2524                        /*
2525                         * This will result in an incorrect ALL_ZONES zoneid
2526                         * for broadcast and multicast packets, but we
2527                         * don't send ICMP errors for those in any case.
2528                         */
2529                        iras.ira_zoneid = ipif_lookup_addr_zoneid(dst,
2530                            ill, ipst);
2531                        ip_drop_input("ICMP_TIME_EXCEEDED reass", mp, ill);
2532                        icmp_time_exceeded(mp,
2533                            ICMP_REASSEMBLY_TIME_EXCEEDED, &iras);
2534                        ASSERT(!(iras.ira_flags & IRAF_IPSEC_SECURE));
2535                }
2536        }
2537        /*
2538         * A non-dying ILL will use the return value to decide whether to
2539         * restart the frag timer, and for how long.
2540         */
2541        return (next_timeout);
2542 }

2544 /*
2545  * This routine is called when the approximate count of mblk memory used
2546  * for the specified ILL has exceeded max_count.
2547  */
2548 void
2549 ill_frag_prune(ill_t *ill, uint_t max_count)
2550 {
2551        ipfb_t  *ipfb;
2552        ipf_t   *ipf;
2553        size_t  count;
2554        clock_t now;

2556        /*
2557         * If we are here within ip_min_frag_prune_time msecs remove
2558         * ill_frag_free_num_pkts oldest packets from each bucket and increment
2559         * ill_frag_free_num_pkts.
2560         */
2561        mutex_enter(&ill->ill_lock);
2562        now = ddi_get_lbolt();
2563        if (TICK_TO_MSEC(now - ill->ill_last_frag_clean_time) <=
2564            (ip_min_frag_prune_time != 0 ?
2565            ip_min_frag_prune_time : msec_per_tick)) {

2567                ill->ill_frag_free_num_pkts++;

2569        } else {
```



```
2570                ill->ill_frag_free_num_pkts = 0;
2571        }
2572        ill->ill_last_frag_clean_time = now;
2573        mutex_exit(&ill->ill_lock);

2575        /*
2576         * free ill_frag_free_num_pkts oldest packets from each bucket.
2577         */
2578        if (ill->ill_frag_free_num_pkts != 0) {
2579                int ix;

2581                for (ix = 0; ix < ILL_FRAG_HASH_TBL_COUNT; ix++) {
2582                        ipfb = &ill->ill_frag_hash_tbl[ix];
2583                        mutex_enter(&ipfb->ipfb_lock);
2584                        if (ipfb->ipfb_ipf != NULL) {
2585                                ill_frag_free_pkts(ill, ipfb, ipfb->ipfb_ipf,
2586                                    ill->ill_frag_free_num_pkts);
2587                        }
2588                        mutex_exit(&ipfb->ipfb_lock);
2589                }
2590        }
2591        /*
2592         * While the reassembly list for this ILL is too big, prune a fragment
2593         * queue by age, oldest first.
2594         */
2595        while (ill->ill_frag_count > max_count) {
2596                int     ix;
2597                ipfb_t  *oipfb = NULL;
2598                uint_t  oldest = UINT_MAX;

2600                count = 0;
2601                for (ix = 0; ix < ILL_FRAG_HASH_TBL_COUNT; ix++) {
2602                        ipfb = &ill->ill_frag_hash_tbl[ix];
2603                        mutex_enter(&ipfb->ipfb_lock);
2604                        ipf = ipfb->ipfb_ipf;
2605                        if (ipf != NULL && ipf->ipf_gen < oldest) {
2606                                oldest = ipf->ipf_gen;
2607                                oipfb = ipfb;
2608                        }
2609                        count += ipfb->ipfb_count;
2610                        mutex_exit(&ipfb->ipfb_lock);
2611                }
2612                if (oipfb == NULL)
2613                        break;

2615                if (count <= max_count)
2616                        return; /* Somebody beat us to it, nothing to do */
2617                mutex_enter(&oipfb->ipfb_lock);
2618                ipf = oipfb->ipfb_ipf;
2619                if (ipf != NULL) {
2620                        ill_frag_free_pkts(ill, oipfb, ipf, 1);
2621                }
2622                mutex_exit(&oipfb->ipfb_lock);
2623        }
2624 }

2626 /*
2627  * free 'free_cnt' fragmented packets starting at ipf.
2628  */
2629 void
2630 ill_frag_free_pkts(ill_t *ill, ipfb_t *ipfb, ipf_t *ipf, int free_cnt)
2631 {
2632        size_t  count;
2633        mblk_t  *mp;
2634        mblk_t  *tmp;
2635        ipf_t **ipfp = ipf->ipf_ptphn;
```

```
2637         ASSERT(MUTEX_HELD(&ipfb->ipfb_lock));
2638         ASSERT(ipfp != NULL);
2639         ASSERT(ipf != NULL);

2641         while (ipf != NULL && free_cnt-- > 0) {
2642                 count = ipf->ipf_count;
2643                 mp = ipf->ipf_mp;
2644                 ipf = ipf->ipf_hash_next;
2645                 for (tmp = mp; tmp; tmp = tmp->b_cont) {
2646                         IP_REASS_SET_START(tmp, 0);
2647                         IP_REASS_SET_END(tmp, 0);
2648                 }
2649                 atomic_add_32(&ill->ill_frag_count, -count);
2650                 ASSERT(ipfb->ipfb_count >= count);
2651                 ipfb->ipfb_count -= count;
2652                 ASSERT(ipfb->ipfb_frag_pkts > 0);
2653                 ipfb->ipfb_frag_pkts--;
2654                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsReasmFails);
2655                 ip_drop_input("ipIfStatsReasmFails", mp, ill);
2656                 freemsg(mp);
2657         }

2659         if (ipf)
2660                 ipf->ipf_ptphn = ipfp;
2661         ipfp[0] = ipf;
2662 }

2664 /*
2665  * Helper function for ill_forward_set().
2666  */
2667 static void
2668 ill_forward_set_on_ill(ill_t *ill, boolean_t enable)
2669 {
2670         ip_stack_t      *ipst = ill->ill_ipst;

2672         ASSERT(IAM_WRITER_ILL(ill) || RW_READ_HELD(&ipst->ips_ill_g_lock));

2674         ip1dbg(("ill_forward_set: %s %s forwarding on %s",
2675             (enable ? "Enabling" : "Disabling"),
2676             (ill->ill_isv6 ? "IPv6" : "IPv4"), ill->ill_name));
2677         mutex_enter(&ill->ill_lock);
2678         if (enable)
2679                 ill->ill_flags |= ILLF_ROUTER;
2680         else
2681                 ill->ill_flags &= ~ILLF_ROUTER;
2682         mutex_exit(&ill->ill_lock);
2683         if (ill->ill_isv6)
2684                 ill_set_nce_router_flags(ill, enable);
2685         /* Notify routing socket listeners of this change. */
2686         if (ill->ill_ipif != NULL)
2687                 ip_rts_ifmsg(ill->ill_ipif, RTSQ_DEFAULT);
2688 }

2690 /*
2691  * Set an ill's ILLF_ROUTER flag appropriately.  Send up RTS_IFINFO routing
2692  * socket messages for each interface whose flags we change.
2693  */
2694 int
2695 ill_forward_set(ill_t *ill, boolean_t enable)
2696 {
2697         ipmp_illgrp_t *illg;
2698         ip_stack_t *ipst = ill->ill_ipst;

2700         ASSERT(IAM_WRITER_ILL(ill) || RW_READ_HELD(&ipst->ips_ill_g_lock));
```



```
2702         if ((enable && (ill->ill_flags & ILLF_ROUTER)) ||
2703             (!enable && !(ill->ill_flags & ILLF_ROUTER)))
2704                 return (0);

2706         if (IS_LOOPBACK(ill))
2707                 return (EINVAL);

2709         if (enable && ill->ill_allowed_ips_cnt > 0)
2710                 return (EPERM);

2712         if (IS_IPMP(ill) || IS_UNDER_IPMP(ill)) {
2713                 /*
2714                  * Update all of the interfaces in the group.
2715                  */
2716                 illg = ill->ill_grp;
2717                 ill = list_head(&illg->ig_if);
2718                 for (; ill != NULL; ill = list_next(&illg->ig_if, ill))
2719                         ill_forward_set_on_ill(ill, enable);

2721                 /*
2722                  * Update the IPMP meta-interface.
2723                  */
2724                 ill_forward_set_on_ill(ipmp_illgrp_ipmp_ill(illg), enable);
2725                 return (0);
2726         }

2728         ill_forward_set_on_ill(ill, enable);
2729         return (0);
2730 }

2732 /*
2733  * Based on the ILLF_ROUTER flag of an ill, make sure all local nce's for
2734  * addresses assigned to the ill have the NCE_F_ISROUTER flag appropriately
2735  * set or clear.
2736  */
2737 static void
2738 ill_set_nce_router_flags(ill_t *ill, boolean_t enable)
2739 {
2740         ipif_t *ipif;
2741         ncec_t *ncec;
2742         nce_t *nce;

2744         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
2745                 /*
2746                  * NOTE: we match across the illgrp because nce's for
2747                  * addresses on IPMP interfaces have an nce_ill that points to
2748                  * the bound underlying ill.
2749                  */
2750                 nce = nce_lookup_v6(ill, &ipif->ipif_v6lcl_addr);
2751                 if (nce != NULL) {
2752                         ncec = nce->nce_common;
2753                         mutex_enter(&ncec->ncec_lock);
2754                         if (enable)
2755                                 ncec->ncec_flags |= NCE_F_ISROUTER;
2756                         else
2757                                 ncec->ncec_flags &= ~NCE_F_ISROUTER;
2758                         mutex_exit(&ncec->ncec_lock);
2759                         nce_refrele(nce);
2760                 }
2761         }
2762 }

2764 /*
2765  * Intializes the context structure and returns the first ill in the list
2766  * cuurently start_list and end_list can have values:
2767  * MAX_G_HEADS          Traverse both IPV4 and IPV6 lists.
```

```
 * IP_V4_G_HEAD         Traverse IPV4 list only.
 * IP_V6_G_HEAD         Traverse IPV6 list only.
 */

/*
 * We don't check for CONDEMNED ills here. Caller must do that if
 * necessary under the ill lock.
 */
ill_t *
ill_first(int start_list, int end_list, ill_walk_context_t *ctx,
    ip_stack_t *ipst)
{
        ill_if_t *ifp;
        ill_t *ill;
        avl_tree_t *avl_tree;

        ASSERT(RW_LOCK_HELD(&ipst->ips_ill_g_lock));
        ASSERT(end_list <= MAX_G_HEADS && start_list >= 0);

        /*
         * setup the lists to search
         */
        if (end_list != MAX_G_HEADS) {
                ctx->ctx_current_list = start_list;
                ctx->ctx_last_list = end_list;
        } else {
                ctx->ctx_last_list = MAX_G_HEADS - 1;
                ctx->ctx_current_list = 0;
        }

        while (ctx->ctx_current_list <= ctx->ctx_last_list) {
                ifp = IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst);
                if (ifp != (ill_if_t *)
                    &IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst)) {
                        avl_tree = &ifp->illif_avl_by_ppa;
                        ill = avl_first(avl_tree);
                        /*
                         * ill is guaranteed to be non NULL or ifp should have
                         * not existed.
                         */
                        ASSERT(ill != NULL);
                        return (ill);
                }
                ctx->ctx_current_list++;
        }

        return (NULL);
}

/*
 * returns the next ill in the list. ill_first() must have been called
 * before calling ill_next() or bad things will happen.
 */

/*
 * We don't check for CONDEMNED ills here. Caller must do that if
 * necessary under the ill lock.
 */
ill_t *
ill_next(ill_walk_context_t *ctx, ill_t *lastill)
{
        ill_if_t *ifp;
        ill_t *ill;
        ip_stack_t      *ipst = lastill->ill_ipst;

        ASSERT(lastill->ill_ifptr != (ill_if_t *)
```



```
            &IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst));
        if ((ill = avl_walk(&lastill->ill_ifptr->illif_avl_by_ppa, lastill,
            AVL_AFTER)) != NULL) {
                return (ill);
        }

        /* goto next ill_ifp in the list. */
        ifp = lastill->ill_ifptr->illif_next;

        /* make sure not at end of circular list */
        while (ifp ==
            (ill_if_t *)&IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst)) {
                if (++ctx->ctx_current_list > ctx->ctx_last_list)
                        return (NULL);
                ifp = IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst);
        }

        return (avl_first(&ifp->illif_avl_by_ppa));
}

/*
 * Check interface name for correct format: [a-zA-Z]+[a-zA-Z0-9._]*[0-9]+
 * The final number (PPA) must not have any leading zeros.  Upon success, a
 * pointer to the start of the PPA is returned; otherwise NULL is returned.
 */
static char *
ill_get_ppa_ptr(char *name)
{
        int namelen = strlen(name);
        int end_ndx = namelen - 1;
        int ppa_ndx, i;

        /*
         * Check that the first character is [a-zA-Z], and that the last
         * character is [0-9].
         */
        if (namelen == 0 || !isalpha(name[0]) || !isdigit(name[end_ndx]))
                return (NULL);

        /*
         * Set 'ppa_ndx' to the PPA start, and check for leading zeroes.
         */
        for (ppa_ndx = end_ndx; ppa_ndx > 0; ppa_ndx--)
                if (!isdigit(name[ppa_ndx - 1]))
                        break;

        if (name[ppa_ndx] == '0' && ppa_ndx < end_ndx)
                return (NULL);

        /*
         * Check that the intermediate characters are [a-z0-9.]
         */
        for (i = 1; i < ppa_ndx; i++) {
                if (!isalpha(name[i]) && !isdigit(name[i]) &&
                    name[i] != '.' && name[i] != '_') {
                        return (NULL);
                }
        }

        return (name + ppa_ndx);
}

/*
 * use avl tree to locate the ill.
 */
static ill_t *
```

```
2900 ill_find_by_name(char *name, boolean_t isv6, ip_stack_t *ipst)
2901 {
2902         char *ppa_ptr = NULL;
2903         int len;
2904         uint_t ppa;
2905         ill_t *ill = NULL;
2906         ill_if_t *ifp;
2907         int list;

2909         /*
2910          * get ppa ptr
2911          */
2912         if (isv6)
2913                 list = IP_V6_G_HEAD;
2914         else
2915                 list = IP_V4_G_HEAD;

2917         if ((ppa_ptr = ill_get_ppa_ptr(name)) == NULL) {
2918                 return (NULL);
2919         }

2921         len = ppa_ptr - name + 1;

2923         ppa = stoi(&ppa_ptr);

2925         ifp = IP_VX_ILL_G_LIST(list, ipst);

2927         while (ifp != (ill_if_t *)&IP_VX_ILL_G_LIST(list, ipst)) {
2928                 /*
2929                  * match is done on len - 1 as the name is not null
2930                  * terminated it contains ppa in addition to the interface
2931                  * name.
2932                  */
2933                 if ((ifp->illif_name_len == len) &&
2934                     bcmp(ifp->illif_name, name, len - 1) == 0) {
2935                         break;
2936                 } else {
2937                         ifp = ifp->illif_next;
2938                 }
2939         }

2941         if (ifp == (ill_if_t *)&IP_VX_ILL_G_LIST(list, ipst)) {
2942                 /*
2943                  * Even the interface type does not exist.
2944                  */
2945                 return (NULL);
2946         }

2948         ill = avl_find(&ifp->illif_avl_by_ppa, (void *) &ppa, NULL);
2949         if (ill != NULL) {
2950                 mutex_enter(&ill->ill_lock);
2951                 if (ILL_CAN_LOOKUP(ill)) {
2952                         ill_refhold_locked(ill);
2953                         mutex_exit(&ill->ill_lock);
2954                         return (ill);
2955                 }
2956                 mutex_exit(&ill->ill_lock);
2957         }
2958         return (NULL);
2959 }

2961 /*
2962  * comparison function for use with avl.
2963  */
2964 static int
2965 ill_compare_ppa(const void *ppa_ptr, const void *ill_ptr)
```



```
2966 {
2967         uint_t ppa;
2968         uint_t ill_ppa;

2970         ASSERT(ppa_ptr != NULL && ill_ptr != NULL);

2972         ppa = *((uint_t *)ppa_ptr);
2973         ill_ppa = ((const ill_t *)ill_ptr)->ill_ppa;
2974         /*
2975          * We want the ill with the lowest ppa to be on the
2976          * top.
2977          */
2978         if (ill_ppa < ppa)
2979                 return (1);
2980         if (ill_ppa > ppa)
2981                 return (-1);
2982         return (0);
2983 }

2985 /*
2986  * remove an interface type from the global list.
2987  */
2988 static void
2989 ill_delete_interface_type(ill_if_t *interface)
2990 {
2991         ASSERT(interface != NULL);
2992         ASSERT(avl_numnodes(&interface->illif_avl_by_ppa) == 0);

2994         avl_destroy(&interface->illif_avl_by_ppa);
2995         if (interface->illif_ppa_arena != NULL)
2996                 vmem_destroy(interface->illif_ppa_arena);

2998         remque(interface);

3000         mi_free(interface);
3001 }

3003 /*
3004  * remove ill from the global list.
3005  */
3006 static void
3007 ill_glist_delete(ill_t *ill)
3008 {
3009         ip_stack_t      *ipst;
3010         phyint_t        *phyi;

3012         if (ill == NULL)
3013                 return;
3014         ipst = ill->ill_ipst;
3015         rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);

3017         /*
3018          * If the ill was never inserted into the AVL tree
3019          * we skip the if branch.
3020          */
3021         if (ill->ill_ifptr != NULL) {
3022                 /*
3023                  * remove from AVL tree and free ppa number
3024                  */
3025                 avl_remove(&ill->ill_ifptr->illif_avl_by_ppa, ill);

3027                 if (ill->ill_ifptr->illif_ppa_arena != NULL) {
3028                         vmem_free(ill->ill_ifptr->illif_ppa_arena,
3029                             (void *)(uintptr_t)(ill->ill_ppa+1), 1);
3030                 }
3031                 if (avl_numnodes(&ill->ill_ifptr->illif_avl_by_ppa) == 0) {
```

```
3032                        ill_delete_interface_type(ill->ill_ifptr);
3033                }
3034
3035                /*
3036                 * Indicate ill is no longer in the list.
3037                 */
3038                ill->ill_ifptr = NULL;
3039                ill->ill_name_length = 0;
3040                ill->ill_name[0] = '\0';
3041                ill->ill_ppa = UINT_MAX;
3042        }
3043
3044        /* Generate one last event for this ill. */
3045        ill_nic_event_dispatch(ill, 0, NE_UNPLUMB, ill->ill_name,
3046            ill->ill_name_length);
3047
3048        ASSERT(ill->ill_phyint != NULL);
3049        phyi = ill->ill_phyint;
3050        ill->ill_phyint = NULL;
3051
3052        /*
3053         * ill_init allocates a phyint always to store the copy
3054         * of flags relevant to phyint. At that point in time, we could
3055         * not assign the name and hence phyint_illv4/v6 could not be
3056         * initialized. Later in ipif_set_values, we assign the name to
3057         * the ill, at which point in time we assign phyint_illv4/v6.
3058         * Thus we don't rely on phyint_illv6 to be initialized always.
3059         */
3060        if (ill->ill_flags & ILLF_IPV6)
3061                phyi->phyint_illv6 = NULL;
3062        else
3063                phyi->phyint_illv4 = NULL;
3064
3065        if (phyi->phyint_illv4 != NULL || phyi->phyint_illv6 != NULL) {
3066                rw_exit(&ipst->ips_ill_g_lock);
3067                return;
3068        }
3069
3070        /*
3071         * There are no ills left on this phyint; pull it out of the phyint
3072         * avl trees, and free it.
3073         */
3074        if (phyi->phyint_ifindex > 0) {
3075                avl_remove(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
3076                    phyi);
3077                avl_remove(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
3078                    phyi);
3079        }
3080        rw_exit(&ipst->ips_ill_g_lock);
3081
3082        phyint_free(phyi);
3083 }
3084
3085 /*
3086  * allocate a ppa, if the number of plumbed interfaces of this type are
3087  * less than ill_no_arena do a linear search to find a unused ppa.
3088  * When the number goes beyond ill_no_arena switch to using an arena.
3089  * Note: ppa value of zero cannot be allocated from vmem_arena as it
3090  * is the return value for an error condition, so allocation starts at one
3091  * and is decremented by one.
3092  */
3093 static int
3094 ill_alloc_ppa(ill_if_t *ifp, ill_t *ill)
3095 {
3096        ill_t *tmp_ill;
3097        uint_t start, end;
```



```
3098        int ppa;
3099
3100        if (ifp->illif_ppa_arena == NULL &&
3101            (avl_numnodes(&ifp->illif_avl_by_ppa) + 1 > ill_no_arena)) {
3102                /*
3103                 * Create an arena.
3104                 */
3105                ifp->illif_ppa_arena = vmem_create(ifp->illif_name,
3106                    (void *)1, UINT_MAX - 1, 1, NULL, NULL,
3107                    NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
3108                        /* allocate what has already been assigned */
3109                for (tmp_ill = avl_first(&ifp->illif_avl_by_ppa);
3110                    tmp_ill != NULL; tmp_ill = avl_walk(&ifp->illif_avl_by_ppa,
3111                    tmp_ill, AVL_AFTER)) {
3112                        ppa = (int)(uintptr_t)vmem_xalloc(ifp->illif_ppa_arena,
3113                            1,          /* size */
3114                            1,          /* align/quantum */
3115                            0,          /* phase */
3116                            0,          /* nocross */
3117                            /* minaddr */
3118                            (void *)((uintptr_t)tmp_ill->ill_ppa + 1),
3119                            /* maxaddr */
3120                            (void *)((uintptr_t)tmp_ill->ill_ppa + 2),
3121                            VM_NOSLEEP|VM_FIRSTFIT);
3122                        if (ppa == 0) {
3123                                ip1dbg(("ill_alloc_ppa: ppa allocation"
3124                                    " failed while switching"));
3125                                vmem_destroy(ifp->illif_ppa_arena);
3126                                ifp->illif_ppa_arena = NULL;
3127                                break;
3128                        }
3129                }
3130        }
3131
3132        if (ifp->illif_ppa_arena != NULL) {
3133                if (ill->ill_ppa == UINT_MAX) {
3134                        ppa = (int)(uintptr_t)vmem_alloc(ifp->illif_ppa_arena,
3135                            1, VM_NOSLEEP|VM_FIRSTFIT);
3136                        if (ppa == 0)
3137                                return (EAGAIN);
3138                        ill->ill_ppa = --ppa;
3139                } else {
3140                        ppa = (int)(uintptr_t)vmem_xalloc(ifp->illif_ppa_arena,
3141                            1,          /* size */
3142                            1,          /* align/quantum */
3143                            0,          /* phase */
3144                            0,          /* nocross */
3145                            (void *)(uintptr_t)(ill->ill_ppa + 1), /* minaddr */
3146                            (void *)(uintptr_t)(ill->ill_ppa + 2), /* maxaddr */
3147                            VM_NOSLEEP|VM_FIRSTFIT);
3148                        /*
3149                         * Most likely the allocation failed because
3150                         * the requested ppa was in use.
3151                         */
3152                        if (ppa == 0)
3153                                return (EEXIST);
3154                }
3155                return (0);
3156        }
3157
3158        /*
3159         * No arena is in use and not enough (>ill_no_arena) interfaces have
3160         * been plumbed to create one. Do a linear search to get a unused ppa.
3161         */
3162        if (ill->ill_ppa == UINT_MAX) {
3163                end = UINT_MAX - 1;
```

```
3164                 start = 0;
3165         } else {
3166                 end = start = ill->ill_ppa;
3167         }

3169         tmp_ill = avl_find(&ifp->illif_avl_by_ppa, (void *)&start, NULL);
3170         while (tmp_ill != NULL && tmp_ill->ill_ppa == start) {
3171                 if (start++ >= end) {
3172                         if (ill->ill_ppa == UINT_MAX)
3173                                 return (EAGAIN);
3174                         else
3175                                 return (EEXIST);
3176                 }
3177                 tmp_ill = avl_walk(&ifp->illif_avl_by_ppa, tmp_ill, AVL_AFTER);
3178         }
3179         ill->ill_ppa = start;
3180         return (0);
3181 }

3183 /*
3184  * Insert ill into the list of configured ill's. Once this function completes,
3185  * the ill is globally visible and is available through lookups. More precisely
3186  * this happens after the caller drops the ill_g_lock.
3187  */
3188 static int
3189 ill_glist_insert(ill_t *ill, char *name, boolean_t isv6)
3190 {
3191         ill_if_t *ill_interface;
3192         avl_index_t where = 0;
3193         int error;
3194         int name_length;
3195         int index;
3196         boolean_t check_length = B_FALSE;
3197         ip_stack_t      *ipst = ill->ill_ipst;

3199         ASSERT(RW_WRITE_HELD(&ipst->ips_ill_g_lock));

3201         name_length = mi_strlen(name) + 1;

3203         if (isv6)
3204                 index = IP_V6_G_HEAD;
3205         else
3206                 index = IP_V4_G_HEAD;

3208         ill_interface = IP_VX_ILL_G_LIST(index, ipst);
3209         /*
3210          * Search for interface type based on name
3211          */
3212         while (ill_interface != (ill_if_t *)&IP_VX_ILL_G_LIST(index, ipst)) {
3213                 if ((ill_interface->illif_name_len == name_length) &&
3214                     (strcmp(ill_interface->illif_name, name) == 0)) {
3215                         break;
3216                 }
3217                 ill_interface = ill_interface->illif_next;
3218         }

3220         /*
3221          * Interface type not found, create one.
3222          */
3223         if (ill_interface == (ill_if_t *)&IP_VX_ILL_G_LIST(index, ipst)) {
3224                 ill_g_head_t ghead;

3226                 /*
3227                  * allocate ill_if_t structure
3228                  */
3229                 ill_interface = (ill_if_t *)mi_zalloc(sizeof (ill_if_t));
```



```
3230                 if (ill_interface == NULL) {
3231                         return (ENOMEM);
3232                 }

3234                 (void) strcpy(ill_interface->illif_name, name);
3235                 ill_interface->illif_name_len = name_length;

3237                 avl_create(&ill_interface->illif_avl_by_ppa,
3238                     ill_compare_ppa, sizeof (ill_t),
3239                     offsetof(struct ill_s, ill_avl_byppa));

3241                 /*
3242                  * link the structure in the back to maintain order
3243                  * of configuration for ifconfig output.
3244                  */
3245                 ghead = ipst->ips_ill_g_heads[index];
3246                 insque(ill_interface, ghead.ill_g_list_tail);
3247         }

3249         if (ill->ill_ppa == UINT_MAX)
3250                 check_length = B_TRUE;

3252         error = ill_alloc_ppa(ill_interface, ill);
3253         if (error != 0) {
3254                 if (avl_numnodes(&ill_interface->illif_avl_by_ppa) == 0)
3255                         ill_delete_interface_type(ill->ill_ifptr);
3256                 return (error);
3257         }

3259         /*
3260          * When the ppa is choosen by the system, check that there is
3261          * enough space to insert ppa. if a specific ppa was passed in this
3262          * check is not required as the interface name passed in will have
3263          * the right ppa in it.
3264          */
3265         if (check_length) {
3266                 /*
3267                  * UINT_MAX - 1 should fit in 10 chars, alloc 12 chars.
3268                  */
3269                 char buf[sizeof (uint_t) * 3];

3271                 /*
3272                  * convert ppa to string to calculate the amount of space
3273                  * required for it in the name.
3274                  */
3275                 numtos(ill->ill_ppa, buf);

3277                 /* Do we have enough space to insert ppa ? */

3279                 if ((mi_strlen(name) + mi_strlen(buf) + 1) > LIFNAMSIZ) {
3280                         /* Free ppa and interface type struct */
3281                         if (ill_interface->illif_ppa_arena != NULL) {
3282                                 vmem_free(ill_interface->illif_ppa_arena,
3283                                     (void *)(uintptr_t)(ill->ill_ppa+1), 1);
3284                         }
3285                         if (avl_numnodes(&ill_interface->illif_avl_by_ppa) == 0)
3286                                 ill_delete_interface_type(ill->ill_ifptr);

3288                         return (EINVAL);
3289                 }
3290         }

3292         (void) sprintf(ill->ill_name, "%s%u", name, ill->ill_ppa);
3293         ill->ill_name_length = mi_strlen(ill->ill_name) + 1;

3295         (void) avl_find(&ill_interface->illif_avl_by_ppa, &ill->ill_ppa,
```

```
3296             &where);
3297         ill->ill_ifptr = ill_interface;
3298         avl_insert(&ill_interface->illif_avl_by_ppa, ill, where);

3300         ill_phyint_reinit(ill);
3301         return (0);
3302 }

3304 /* Initialize the per phyint ipsq used for serialization */
3305 static boolean_t
3306 ipsq_init(ill_t *ill, boolean_t enter)
3307 {
3308         ipsq_t  *ipsq;
3309         ipxop_t *ipx;

3311         if ((ipsq = kmem_zalloc(sizeof (ipsq_t), KM_NOSLEEP)) == NULL)
3312                 return (B_FALSE);

3314         ill->ill_phyint->phyint_ipsq = ipsq;
3315         ipx = ipsq->ipsq_xop = &ipsq->ipsq_ownxop;
3316         ipx->ipx_ipsq = ipsq;
3317         ipsq->ipsq_next = ipsq;
3318         ipsq->ipsq_phyint = ill->ill_phyint;
3319         mutex_init(&ipsq->ipsq_lock, NULL, MUTEX_DEFAULT, 0);
3320         mutex_init(&ipx->ipx_lock, NULL, MUTEX_DEFAULT, 0);
3321         ipsq->ipsq_ipst = ill->ill_ipst;        /* No netstack_hold */
3322         if (enter) {
3323                 ipx->ipx_writer = curthread;
3324                 ipx->ipx_forced = B_FALSE;
3325                 ipx->ipx_reentry_cnt = 1;
3326 #ifdef DEBUG
3327                 ipx->ipx_depth = getpcstack(ipx->ipx_stack, IPX_STACK_DEPTH);
3328 #endif
3329         }
3330         return (B_TRUE);
3331 }

3333 /*
3334  * ill_init is called by ip_open when a device control stream is opened.
3335  * It does a few initializations, and shoots a DL_INFO_REQ message down
3336  * to the driver.  The response is later picked up in ip_rput_dlpi and
3337  * used to set up default mechanisms for talking to the driver.  (Always
3338  * called as writer.)
3339  *
3340  * If this function returns error, ip_open will call ip_close which in
3341  * turn will call ill_delete to clean up any memory allocated here that
3342  * is not yet freed.
3343  */
3344 int
3345 ill_init(queue_t *q, ill_t *ill)
3346 {
3347         int     count;
3348         dl_info_req_t   *dlir;
3349         mblk_t  *info_mp;
3350         uchar_t *frag_ptr;

3352         /*
3353          * The ill is initialized to zero by mi_alloc*(). In addition
3354          * some fields already contain valid values, initialized in
3355          * ip_open(), before we reach here.
3356          */
3357         mutex_init(&ill->ill_lock, NULL, MUTEX_DEFAULT, 0);
3358         mutex_init(&ill->ill_saved_ire_lock, NULL, MUTEX_DEFAULT, NULL);
3359         ill->ill_saved_ire_cnt = 0;

3361         ill->ill_rq = q;
```



```
3362         ill->ill_wq = WR(q);

3364         info_mp = allocb(MAX(sizeof (dl_info_req_t), sizeof (dl_info_ack_t)),
3365             BPRI_HI);
3366         if (info_mp == NULL)
3367                 return (ENOMEM);

3369         /*
3370          * Allocate sufficient space to contain our fragment hash table and
3371          * the device name.
3372          */
3373         frag_ptr = (uchar_t *)mi_zalloc(ILL_FRAG_HASH_TBL_SIZE + 2 * LIFNAMSIZ);
3374         if (frag_ptr == NULL) {
3375                 freemsg(info_mp);
3376                 return (ENOMEM);
3377         }
3378         ill->ill_frag_ptr = frag_ptr;
3379         ill->ill_frag_free_num_pkts = 0;
3380         ill->ill_last_frag_clean_time = 0;
3381         ill->ill_frag_hash_tbl = (ipfb_t *)frag_ptr;
3382         ill->ill_name = (char *)(frag_ptr + ILL_FRAG_HASH_TBL_SIZE);
3383         for (count = 0; count < ILL_FRAG_HASH_TBL_COUNT; count++) {
3384                 mutex_init(&ill->ill_frag_hash_tbl[count].ipfb_lock,
3385                     NULL, MUTEX_DEFAULT, NULL);
3386         }

3388         ill->ill_phyint = (phyint_t *)mi_zalloc(sizeof (phyint_t));
3389         if (ill->ill_phyint == NULL) {
3390                 freemsg(info_mp);
3391                 mi_free(frag_ptr);
3392                 return (ENOMEM);
3393         }

3395         mutex_init(&ill->ill_phyint->phyint_lock, NULL, MUTEX_DEFAULT, 0);
3396         /*
3397          * For now pretend this is a v4 ill. We need to set phyint_ill*
3398          * at this point because of the following reason. If we can't
3399          * enter the ipsq at some point and cv_wait, the writer that
3400          * wakes us up tries to locate us using the list of all phyints
3401          * in an ipsq and the ills from the phyint thru the phyint_ill*.
3402          * If we don't set it now, we risk a missed wakeup.
3403          */
3404         ill->ill_phyint->phyint_illv4 = ill;
3405         ill->ill_ppa = UINT_MAX;
3406         list_create(&ill->ill_nce, sizeof (nce_t), offsetof(nce_t, nce_node));

3408         ill_set_inputfn(ill);

3410         if (!ipsq_init(ill, B_TRUE)) {
3411                 freemsg(info_mp);
3412                 mi_free(frag_ptr);
3413                 mi_free(ill->ill_phyint);
3414                 return (ENOMEM);
3415         }

3417         ill->ill_state_flags |= ILL_LL_SUBNET_PENDING;

3419         /* Frag queue limit stuff */
3420         ill->ill_frag_count = 0;
3421         ill->ill_ipf_gen = 0;

3423         rw_init(&ill->ill_mcast_lock, NULL, RW_DEFAULT, NULL);
3424         mutex_init(&ill->ill_mcast_serializer, NULL, MUTEX_DEFAULT, NULL);
3425         ill->ill_global_timer = INFINITY;
3426         ill->ill_mcast_v1_time = ill->ill_mcast_v2_time = 0;
3427         ill->ill_mcast_v1_tset = ill->ill_mcast_v2_tset = 0;
```

```
3428         ill->ill_mcast_rv = MCAST_DEF_ROBUSTNESS;
3429         ill->ill_mcast_qi = MCAST_DEF_QUERY_INTERVAL;

3431         /*
3432          * Initialize IPv6 configuration variables.  The IP module is always
3433          * opened as an IPv4 module.  Instead tracking down the cases where
3434          * it switches to do ipv6, we'll just initialize the IPv6 configuration
3435          * here for convenience, this has no effect until the ill is set to do
3436          * IPv6.
3437          */
3438         ill->ill_reachable_time = ND_REACHABLE_TIME;
3439         ill->ill_xmit_count = ND_MAX_MULTICAST_SOLICIT;
3440         ill->ill_max_buf = ND_MAX_Q;
3441         ill->ill_refcnt = 0;

3443         /* Send down the Info Request to the driver. */
3444         info_mp->b_datap->db_type = M_PCPROTO;
3445         dlir = (dl_info_req_t *)info_mp->b_rptr;
3446         info_mp->b_wptr = (uchar_t *)&dlir[1];
3447         dlir->dl_primitive = DL_INFO_REQ;

3449         ill->ill_dlpi_pending = DL_PRIM_INVAL;

3451         qprocson(q);
3452         ill_dlpi_send(ill, info_mp);

3454         return (0);
3455 }

3457 /*
3458  * ill_dls_info
3459  * creates datalink socket info from the device.
3460  */
3461 int
3462 ill_dls_info(struct sockaddr_dl *sdl, const ill_t *ill)
3463 {
3464         size_t  len;

3466         sdl->sdl_family = AF_LINK;
3467         sdl->sdl_index = ill_get_upper_ifindex(ill);
3468         sdl->sdl_type = ill->ill_type;
3469         ill_get_name(ill, sdl->sdl_data, sizeof (sdl->sdl_data));
3470         len = strlen(sdl->sdl_data);
3471         ASSERT(len < 256);
3472         sdl->sdl_nlen = (uchar_t)len;
3473         sdl->sdl_alen = ill->ill_phys_addr_length;
3474         sdl->sdl_slen = 0;
3475         if (ill->ill_phys_addr_length != 0 && ill->ill_phys_addr != NULL)
3476                 bcopy(ill->ill_phys_addr, &sdl->sdl_data[len], sdl->sdl_alen);

3478         return (sizeof (struct sockaddr_dl));
3479 }

3481 /*
3482  * ill_xarp_info
3483  * creates xarp info from the device.
3484  */
3485 static int
3486 ill_xarp_info(struct sockaddr_dl *sdl, ill_t *ill)
3487 {
3488         sdl->sdl_family = AF_LINK;
3489         sdl->sdl_index = ill->ill_phyint->phyint_ifindex;
3490         sdl->sdl_type = ill->ill_type;
3491         ill_get_name(ill, sdl->sdl_data, sizeof (sdl->sdl_data));
3492         sdl->sdl_nlen = (uchar_t)mi_strlen(sdl->sdl_data);
3493         sdl->sdl_alen = ill->ill_phys_addr_length;
```



```
3494         sdl->sdl_slen = 0;
3495         return (sdl->sdl_nlen);
3496 }

3498 static int
3499 loopback_kstat_update(kstat_t *ksp, int rw)
3500 {
3501         kstat_named_t *kn;
3502         netstackid_t    stackid;
3503         netstack_t      *ns;
3504         ip_stack_t      *ipst;

3506         if (ksp == NULL || ksp->ks_data == NULL)
3507                 return (EIO);

3509         if (rw == KSTAT_WRITE)
3510                 return (EACCES);

3512         kn = KSTAT_NAMED_PTR(ksp);
3513         stackid = (zoneid_t)(uintptr_t)ksp->ks_private;

3515         ns = netstack_find_by_stackid(stackid);
3516         if (ns == NULL)
3517                 return (-1);

3519         ipst = ns->netstack_ip;
3520         if (ipst == NULL) {
3521                 netstack_rele(ns);
3522                 return (-1);
3523         }
3524         kn[0].value.ui32 = ipst->ips_loopback_packets;
3525         kn[1].value.ui32 = ipst->ips_loopback_packets;
3526         netstack_rele(ns);
3527         return (0);
3528 }

3530 /*
3531  * Has ifindex been plumbed already?
3532  */
3533 static boolean_t
3534 phyint_exists(uint_t index, ip_stack_t *ipst)
3535 {
3536         ASSERT(index != 0);
3537         ASSERT(RW_LOCK_HELD(&ipst->ips_ill_g_lock));

3539         return (avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
3540             &index, NULL) != NULL);
3541 }

3543 /*
3544  * Pick a unique ifindex.
3545  * When the index counter passes IF_INDEX_MAX for the first time, the wrap
3546  * flag is set so that next time time ip_assign_ifindex() is called, it
3547  * falls through and resets the index counter back to 1, the minimum value
3548  * for the interface index. The logic below assumes that ips_ill_index
3549  * can hold a value of IF_INDEX_MAX+1 without there being any loss
3550  * (i.e. reset back to 0.)
3551  */
3552 boolean_t
3553 ip_assign_ifindex(uint_t *indexp, ip_stack_t *ipst)
3554 {
3555         uint_t loops;

3557         if (!ipst->ips_ill_index_wrap) {
3558                 *indexp = ipst->ips_ill_index++;
3559                 if (ipst->ips_ill_index > IF_INDEX_MAX) {
```

```
3560                        /*
3561                         * Reached the maximum ifindex value, set the wrap
3562                         * flag to indicate that it is no longer possible
3563                         * to assume that a given index is unallocated.
3564                         */
3565                        ipst->ips_ill_index_wrap = B_TRUE;
3566                }
3567                return (B_TRUE);
3568        }

3570        if (ipst->ips_ill_index > IF_INDEX_MAX)
3571                ipst->ips_ill_index = 1;

3573        /*
3574         * Start reusing unused indexes. Note that we hold the ill_g_lock
3575         * at this point and don't want to call any function that attempts
3576         * to get the lock again.
3577         */
3578        for (loops = IF_INDEX_MAX; loops > 0; loops--) {
3579                if (!phyint_exists(ipst->ips_ill_index, ipst)) {
3580                        /* found unused index - use it */
3581                        *indexp = ipst->ips_ill_index;
3582                        return (B_TRUE);
3583                }

3585                ipst->ips_ill_index++;
3586                if (ipst->ips_ill_index > IF_INDEX_MAX)
3587                        ipst->ips_ill_index = 1;
3588        }

3590        /*
3591         * all interface indicies are inuse.
3592         */
3593        return (B_FALSE);
3594 }

3596 /*
3597  * Assign a unique interface index for the phyint.
3598  */
3599 static boolean_t
3600 phyint_assign_ifindex(phyint_t *phyi, ip_stack_t *ipst)
3601 {
3602        ASSERT(phyi->phyint_ifindex == 0);
3603        return (ip_assign_ifindex(&phyi->phyint_ifindex, ipst));
3604 }

3606 /*
3607  * Initialize the flags on 'phyi' as per the provided mactype.
3608  */
3609 static void
3610 phyint_flags_init(phyint_t *phyi, t_uscalar_t mactype)
3611 {
3612        uint64_t flags = 0;

3614        /*
3615         * Initialize PHYI_RUNNING and PHYI_FAILED.  For non-IPMP interfaces,
3616         * we always presume the underlying hardware is working and set
3617         * PHYI_RUNNING (if it's not, the driver will subsequently send a
3618         * DL_NOTE_LINK_DOWN message).  For IPMP interfaces, at initialization
3619         * there are no active interfaces in the group so we set PHYI_FAILED.
3620         */
3621        if (mactype == SUNW_DL_IPMP)
3622                flags |= PHYI_FAILED;
3623        else
3624                flags |= PHYI_RUNNING;
```



```
3626        switch (mactype) {
3627        case SUNW_DL_VNI:
3628                flags |= PHYI_VIRTUAL;
3629                break;
3630        case SUNW_DL_IPMP:
3631                flags |= PHYI_IPMP;
3632                break;
3633        case DL_LOOP:
3634                flags |= (PHYI_LOOPBACK | PHYI_VIRTUAL);
3635                break;
3636        }

3638        mutex_enter(&phyi->phyint_lock);
3639        phyi->phyint_flags |= flags;
3640        mutex_exit(&phyi->phyint_lock);
3641 }

3643 /*
3644  * Return a pointer to the ill which matches the supplied name.  Note that
3645  * the ill name length includes the null termination character.  (May be
3646  * called as writer.)
3647  * If do_alloc and the interface is "lo0" it will be automatically created.
3648  * Cannot bump up reference on condemned ills. So dup detect can't be done
3649  * using this func.
3650  */
3651 ill_t *
3652 ill_lookup_on_name(char *name, boolean_t do_alloc, boolean_t isv6,
3653     boolean_t *did_alloc, ip_stack_t *ipst)
3654 {
3655        ill_t   *ill;
3656        ipif_t  *ipif;
3657        ipsq_t  *ipsq;
3658        kstat_named_t   *kn;
3659        boolean_t isloopback;
3660        in6_addr_t ov6addr;

3662        isloopback = mi_strcmp(name, ipif_loopback_name) == 0;

3664        rw_enter(&ipst->ips_ill_g_lock, RW_READER);
3665        ill = ill_find_by_name(name, isv6, ipst);
3666        rw_exit(&ipst->ips_ill_g_lock);
3667        if (ill != NULL)
3668                return (ill);

3670        /*
3671         * Couldn't find it.  Does this happen to be a lookup for the
3672         * loopback device and are we allowed to allocate it?
3673         */
3674        if (!isloopback || !do_alloc)
3675                return (NULL);

3677        rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
3678        ill = ill_find_by_name(name, isv6, ipst);
3679        if (ill != NULL) {
3680                rw_exit(&ipst->ips_ill_g_lock);
3681                return (ill);
3682        }

3684        /* Create the loopback device on demand */
3685        ill = (ill_t *)(mi_alloc(sizeof (ill_t) +
3686            sizeof (ipif_loopback_name), BPRI_MED));
3687        if (ill == NULL)
3688                goto done;

3690        *ill = ill_null;
3691        mutex_init(&ill->ill_lock, NULL, MUTEX_DEFAULT, NULL);
```

```
3692         ill->ill_ipst = ipst;
3693         list_create(&ill->ill_nce, sizeof (nce_t), offsetof(nce_t, nce_node));
3694         netstack_hold(ipst->ips_netstack);
3695         /*
3696          * For exclusive stacks we set the zoneid to zero
3697          * to make IP operate as if in the global zone.
3698          */
3699         ill->ill_zoneid = GLOBAL_ZONEID;

3701         ill->ill_phyint = (phyint_t *)mi_zalloc(sizeof (phyint_t));
3702         if (ill->ill_phyint == NULL)
3703                 goto done;

3705         if (isv6)
3706                 ill->ill_phyint->phyint_illv6 = ill;
3707         else
3708                 ill->ill_phyint->phyint_illv4 = ill;
3709         mutex_init(&ill->ill_phyint->phyint_lock, NULL, MUTEX_DEFAULT, 0);
3710         phyint_flags_init(ill->ill_phyint, DL_LOOP);

3712         if (isv6) {
3713                 ill->ill_isv6 = B_TRUE;
3714                 ill->ill_max_frag = ip_loopback_mtu_v6plus;
3715         } else {
3716                 ill->ill_max_frag = ip_loopback_mtuplus;
3717         }
3718         if (!ill_allocate_mibs(ill))
3719                 goto done;
3720         ill->ill_current_frag = ill->ill_max_frag;
3721         ill->ill_mtu = ill->ill_max_frag;       /* Initial value */
3722         ill->ill_mc_mtu = ill->ill_mtu;
3723         /*
3724          * ipif_loopback_name can't be pointed at directly because its used
3725          * by both the ipv4 and ipv6 interfaces.  When the ill is removed
3726          * from the glist, ill_glist_delete() sets the first character of
3727          * ill_name to '\0'.
3728          */
3729         ill->ill_name = (char *)ill + sizeof (*ill);
3730         (void) strcpy(ill->ill_name, ipif_loopback_name);
3731         ill->ill_name_length = sizeof (ipif_loopback_name);
3732         /* Set ill_dlpi_pending for ipsq_current_finish() to work properly */
3733         ill->ill_dlpi_pending = DL_PRIM_INVAL;

3735         rw_init(&ill->ill_mcast_lock, NULL, RW_DEFAULT, NULL);
3736         mutex_init(&ill->ill_mcast_serializer, NULL, MUTEX_DEFAULT, NULL);
3737         ill->ill_global_timer = INFINITY;
3738         ill->ill_mcast_v1_time = ill->ill_mcast_v2_time = 0;
3739         ill->ill_mcast_v1_tset = ill->ill_mcast_v2_tset = 0;
3740         ill->ill_mcast_rv = MCAST_DEF_ROBUSTNESS;
3741         ill->ill_mcast_qi = MCAST_DEF_QUERY_INTERVAL;

3743         /* No resolver here. */
3744         ill->ill_net_type = IRE_LOOPBACK;

3746         /* Initialize the ipsq */
3747         if (!ipsq_init(ill, B_FALSE))
3748                 goto done;

3750         ipif = ipif_allocate(ill, 0L, IRE_LOOPBACK, B_TRUE, B_TRUE, NULL);
3751         if (ipif == NULL)
3752                 goto done;

3754         ill->ill_flags = ILLF_MULTICAST;

3756         ov6addr = ipif->ipif_v6lcl_addr;
3757         /* Set up default loopback address and mask. */
```



```
3758         if (!isv6) {
3759                 ipaddr_t inaddr_loopback = htonl(INADDR_LOOPBACK);

3761                 IN6_IPADDR_TO_V4MAPPED(inaddr_loopback, &ipif->ipif_v6lcl_addr);
3762                 V4MASK_TO_V6(htonl(IN_CLASSA_NET), ipif->ipif_v6net_mask);
3763                 V6_MASK_COPY(ipif->ipif_v6lcl_addr, ipif->ipif_v6net_mask,
3764                     ipif->ipif_v6subnet);
3765                 ill->ill_flags |= ILLF_IPV4;
3766         } else {
3767                 ipif->ipif_v6lcl_addr = ipv6_loopback;
3768                 ipif->ipif_v6net_mask = ipv6_all_ones;
3769                 V6_MASK_COPY(ipif->ipif_v6lcl_addr, ipif->ipif_v6net_mask,
3770                     ipif->ipif_v6subnet);
3771                 ill->ill_flags |= ILLF_IPV6;
3772         }

3774         /*
3775          * Chain us in at the end of the ill list. hold the ill
3776          * before we make it globally visible. 1 for the lookup.
3777          */
3778         ill->ill_refcnt = 0;
3779         ill_refhold(ill);

3781         ill->ill_frag_count = 0;
3782         ill->ill_frag_free_num_pkts = 0;
3783         ill->ill_last_frag_clean_time = 0;

3785         ipsq = ill->ill_phyint->phyint_ipsq;

3787         ill_set_inputfn(ill);

3789         if (ill_glist_insert(ill, "lo", isv6) != 0)
3790                 cmn_err(CE_PANIC, "cannot insert loopback interface");

3792         /* Let SCTP know so that it can add this to its list */
3793         sctp_update_ill(ill, SCTP_ILL_INSERT);

3795         /*
3796          * We have already assigned ipif_v6lcl_addr above, but we need to
3797          * call sctp_update_ipif_addr() after SCTP_ILL_INSERT, which
3798          * requires to be after ill_glist_insert() since we need the
3799          * ill_index set. Pass on ipv6_loopback as the old address.
3800          */
3801         sctp_update_ipif_addr(ipif, ov6addr);

3803         ip_rts_newaddrmsg(RTM_CHGADDR, 0, ipif, RTSQ_DEFAULT);

3805         /*
3806          * ill_glist_insert() -> ill_phyint_reinit() may have merged IPSQs.
3807          * If so, free our original one.
3808          */
3809         if (ipsq != ill->ill_phyint->phyint_ipsq)
3810                 ipsq_delete(ipsq);

3812         if (ipst->ips_loopback_ksp == NULL) {
3813                 /* Export loopback interface statistics */
3814                 ipst->ips_loopback_ksp = kstat_create_netstack("lo", 0,
3815                     ipif_loopback_name, "net",
3816                     KSTAT_TYPE_NAMED, 2, 0,
3817                     ipst->ips_netstack->netstack_stackid);
3818                 if (ipst->ips_loopback_ksp != NULL) {
3819                         ipst->ips_loopback_ksp->ks_update =
3820                             loopback_kstat_update;
3821                         kn = KSTAT_NAMED_PTR(ipst->ips_loopback_ksp);
3822                         kstat_named_init(&kn[0], "ipackets", KSTAT_DATA_UINT32);
3823                         kstat_named_init(&kn[1], "opackets", KSTAT_DATA_UINT32);
```

```
3824                        ipst->ips_loopback_ksp->ks_private =
3825                            (void *)(uintptr_t)ipst->ips_netstack->
3826                            netstack_stackid;
3827                        kstat_install(ipst->ips_loopback_ksp);
3828                }
3829        }

3831        *did_alloc = B_TRUE;
3832        rw_exit(&ipst->ips_ill_g_lock);
3833        ill_nic_event_dispatch(ill, MAP_IPIF_ID(ill->ill_ipif->ipif_id),
3834            NE_PLUMB, ill->ill_name, ill->ill_name_length);
3835        return (ill);
3836 done:
3837        if (ill != NULL) {
3838                if (ill->ill_phyint != NULL) {
3839                        ipsq = ill->ill_phyint->phyint_ipsq;
3840                        if (ipsq != NULL) {
3841                                ipsq->ipsq_phyint = NULL;
3842                                ipsq_delete(ipsq);
3843                        }
3844                        mi_free(ill->ill_phyint);
3845                }
3846                ill_free_mib(ill);
3847                if (ill->ill_ipst != NULL)
3848                        netstack_rele(ill->ill_ipst->ips_netstack);
3849                mi_free(ill);
3850        }
3851        rw_exit(&ipst->ips_ill_g_lock);
3852        return (NULL);
3853 }

3855 /*
3856  * For IPP calls - use the ip_stack_t for global stack.
3857  */
3858 ill_t *
3859 ill_lookup_on_ifindex_global_instance(uint_t index, boolean_t isv6)
3860 {
3861        ip_stack_t      *ipst;
3862        ill_t           *ill;

3864        ipst = netstack_find_by_stackid(GLOBAL_NETSTACKID)->netstack_ip;
3865        if (ipst == NULL) {
3866                cmn_err(CE_WARN, "No ip_stack_t for zoneid zero!\n");
3867                return (NULL);
3868        }

3870        ill = ill_lookup_on_ifindex(index, isv6, ipst);
3871        netstack_rele(ipst->ips_netstack);
3872        return (ill);
3873 }

3875 /*
3876  * Return a pointer to the ill which matches the index and IP version type.
3877  */
3878 ill_t *
3879 ill_lookup_on_ifindex(uint_t index, boolean_t isv6, ip_stack_t *ipst)
3880 {
3881        ill_t   *ill;
3882        phyint_t *phyi;

3884        /*
3885         * Indexes are stored in the phyint - a common structure
3886         * to both IPv4 and IPv6.
3887         */
3888        rw_enter(&ipst->ips_ill_g_lock, RW_READER);
3889        phyi = avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
```



```
3890            (void *) &index, NULL);
3891        if (phyi != NULL) {
3892                ill = isv6 ? phyi->phyint_illv6: phyi->phyint_illv4;
3893                if (ill != NULL) {
3894                        mutex_enter(&ill->ill_lock);
3895                        if (!ILL_IS_CONDEMNED(ill)) {
3896                                ill_refhold_locked(ill);
3897                                mutex_exit(&ill->ill_lock);
3898                                rw_exit(&ipst->ips_ill_g_lock);
3899                                return (ill);
3900                        }
3901                        mutex_exit(&ill->ill_lock);
3902                }
3903        }
3904        rw_exit(&ipst->ips_ill_g_lock);
3905        return (NULL);
3906 }

3908 /*
3909  * Verify whether or not an interface index is valid for the specified zoneid
3910  * to transmit packets.
3911  * It can be zero (meaning "reset") or an interface index assigned
3912  * to a non-VNI interface. (We don't use VNI interface to send packets.)
3913  */
3914 boolean_t
3915 ip_xmit_ifindex_valid(uint_t ifindex, zoneid_t zoneid, boolean_t isv6,
3916     ip_stack_t *ipst)
3917 {
3918        ill_t           *ill;

3920        if (ifindex == 0)
3921                return (B_TRUE);

3923        ill = ill_lookup_on_ifindex_zoneid(ifindex, zoneid, isv6, ipst);
3924        if (ill == NULL)
3925                return (B_FALSE);
3926        if (IS_VNI(ill)) {
3927                ill_refrele(ill);
3928                return (B_FALSE);
3929        }
3930        ill_refrele(ill);
3931        return (B_TRUE);
3932 }

3934 /*
3935  * Return the ifindex next in sequence after the passed in ifindex.
3936  * If there is no next ifindex for the given protocol, return 0.
3937  */
3938 uint_t
3939 ill_get_next_ifindex(uint_t index, boolean_t isv6, ip_stack_t *ipst)
3940 {
3941        phyint_t *phyi;
3942        phyint_t *phyi_initial;
3943        uint_t   ifindex;

3945        rw_enter(&ipst->ips_ill_g_lock, RW_READER);

3947        if (index == 0) {
3948                phyi = avl_first(
3949                    &ipst->ips_phyint_g_list->phyint_list_avl_by_index);
3950        } else {
3951                phyi = phyi_initial = avl_find(
3952                    &ipst->ips_phyint_g_list->phyint_list_avl_by_index,
3953                    (void *) &index, NULL);
3954        }
```

```
3956         for (; phyi != NULL;
3957             phyi = avl_walk(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
3958             phyi, AVL_AFTER)) {
3959                 /*
3960                  * If we're not returning the first interface in the tree
3961                  * and we still haven't moved past the phyint_t that
3962                  * corresponds to index, avl_walk needs to be called again
3963                  */
3964                 if (!((index != 0) && (phyi == phyi_initial))) {
3965                         if (isv6) {
3966                                 if ((phyi->phyint_illv6) &&
3967                                     ILL_CAN_LOOKUP(phyi->phyint_illv6) &&
3968                                     (phyi->phyint_illv6->ill_isv6 == 1))
3969                                         break;
3970                         } else {
3971                                 if ((phyi->phyint_illv4) &&
3972                                     ILL_CAN_LOOKUP(phyi->phyint_illv4) &&
3973                                     (phyi->phyint_illv4->ill_isv6 == 0))
3974                                         break;
3975                         }
3976                 }
3977         }

3979         rw_exit(&ipst->ips_ill_g_lock);

3981         if (phyi != NULL)
3982                 ifindex = phyi->phyint_ifindex;
3983         else
3984                 ifindex = 0;

3986         return (ifindex);
3987 }

3989 /*
3990  * Return the ifindex for the named interface.
3991  * If there is no next ifindex for the interface, return 0.
3992  */
3993 uint_t
3994 ill_get_ifindex_by_name(char *name, ip_stack_t *ipst)
3995 {
3996         phyint_t        *phyi;
3997         avl_index_t     where = 0;
3998         uint_t          ifindex;

4000         rw_enter(&ipst->ips_ill_g_lock, RW_READER);

4002         if ((phyi = avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
4003             name, &where)) == NULL) {
4004                 rw_exit(&ipst->ips_ill_g_lock);
4005                 return (0);
4006         }

4008         ifindex = phyi->phyint_ifindex;

4010         rw_exit(&ipst->ips_ill_g_lock);

4012         return (ifindex);
4013 }

4015 /*
4016  * Return the ifindex to be used by upper layer protocols for instance
4017  * for IPV6_RECVPKTINFO. If IPMP this is the one for the upper ill.
4018  */
4019 uint_t
4020 ill_get_upper_ifindex(const ill_t *ill)
4021 {
```



```
4022         if (IS_UNDER_IPMP(ill))
4023                 return (ipmp_ill_get_ipmp_ifindex(ill));
4024         else
4025                 return (ill->ill_phyint->phyint_ifindex);
4026 }


4029 /*
4030  * Obtain a reference to the ill. The ill_refcnt is a dynamic refcnt
4031  * that gives a running thread a reference to the ill. This reference must be
4032  * released by the thread when it is done accessing the ill and related
4033  * objects. ill_refcnt can not be used to account for static references
4034  * such as other structures pointing to an ill. Callers must generally
4035  * check whether an ill can be refheld by using ILL_CAN_LOOKUP macros
4036  * or be sure that the ill is not being deleted or changing state before
4037  * calling the refhold functions. A non-zero ill_refcnt ensures that the
4038  * ill won't change any of its critical state such as address, netmask etc.
4039  */
4040 void
4041 ill_refhold(ill_t *ill)
4042 {
4043         mutex_enter(&ill->ill_lock);
4044         ill->ill_refcnt++;
4045         ILL_TRACE_REF(ill);
4046         mutex_exit(&ill->ill_lock);
4047 }

4049 void
4050 ill_refhold_locked(ill_t *ill)
4051 {
4052         ASSERT(MUTEX_HELD(&ill->ill_lock));
4053         ill->ill_refcnt++;
4054         ILL_TRACE_REF(ill);
4055 }

4057 /* Returns true if we managed to get a refhold */
4058 boolean_t
4059 ill_check_and_refhold(ill_t *ill)
4060 {
4061         mutex_enter(&ill->ill_lock);
4062         if (!ILL_IS_CONDEMNED(ill)) {
4063                 ill_refhold_locked(ill);
4064                 mutex_exit(&ill->ill_lock);
4065                 return (B_TRUE);
4066         }
4067         mutex_exit(&ill->ill_lock);
4068         return (B_FALSE);
4069 }

4071 /*
4072  * Must not be called while holding any locks. Otherwise if this is
4073  * the last reference to be released, there is a chance of recursive mutex
4074  * panic due to ill_refrele -> ipif_ill_refrele_tail -> qwriter_ip trying
4075  * to restart an ioctl.
4076  */
4077 void
4078 ill_refrele(ill_t *ill)
4079 {
4080         mutex_enter(&ill->ill_lock);
4081         ASSERT(ill->ill_refcnt != 0);
4082         ill->ill_refcnt--;
4083         ILL_UNTRACE_REF(ill);
4084         if (ill->ill_refcnt != 0) {
4085                 /* Every ire pointing to the ill adds 1 to ill_refcnt */
4086                 mutex_exit(&ill->ill_lock);
4087                 return;
```

```
4088         }

4090         /* Drops the ill_lock */
4091         ipif_ill_refrele_tail(ill);
4092 }

4094 /*
4095  * Obtain a weak reference count on the ill. This reference ensures the
4096  * ill won't be freed, but the ill may change any of its critical state
4097  * such as netmask, address etc. Returns an error if the ill has started
4098  * closing.
4099  */
4100 boolean_t
4101 ill_waiter_inc(ill_t *ill)
4102 {
4103         mutex_enter(&ill->ill_lock);
4104         if (ill->ill_state_flags & ILL_CONDEMNED) {
4105                 mutex_exit(&ill->ill_lock);
4106                 return (B_FALSE);
4107         }
4108         ill->ill_waiters++;
4109         mutex_exit(&ill->ill_lock);
4110         return (B_TRUE);
4111 }

4113 void
4114 ill_waiter_dcr(ill_t *ill)
4115 {
4116         mutex_enter(&ill->ill_lock);
4117         ill->ill_waiters--;
4118         if (ill->ill_waiters == 0)
4119                 cv_broadcast(&ill->ill_cv);
4120         mutex_exit(&ill->ill_lock);
4121 }

4123 /*
4124  * ip_ll_subnet_defaults is called when we get the DL_INFO_ACK back from the
4125  * driver.  We construct best guess defaults for lower level information that
4126  * we need.  If an interface is brought up without injection of any overriding
4127  * information from outside, we have to be ready to go with these defaults.
4128  * When we get the first DL_INFO_ACK (from ip_open() sending a DL_INFO_REQ)
4129  * we primarely want the dl_provider_style.
4130  * The subsequent DL_INFO_ACK is received after doing a DL_ATTACH and DL_BIND
4131  * at which point we assume the other part of the information is valid.
4132  */
4133 void
4134 ip_ll_subnet_defaults(ill_t *ill, mblk_t *mp)
4135 {
4136         uchar_t         *brdcst_addr;
4137         uint_t          brdcst_addr_length, phys_addr_length;
4138         t_scalar_t      sap_length;
4139         dl_info_ack_t   *dlia;
4140         ip_m_t          *ipm;
4141         dl_qos_cl_sel1_t *sel1;
4142         int             min_mtu;

4144         ASSERT(IAM_WRITER_ILL(ill));

4146         /*
4147          * Till the ill is fully up  the ill is not globally visible.
4148          * So no need for a lock.
4149          */
4150         dlia = (dl_info_ack_t *)mp->b_rptr;
4151         ill->ill_mactype = dlia->dl_mac_type;

4153         ipm = ip_m_lookup(dlia->dl_mac_type);
```



```
4154         if (ipm == NULL) {
4155                 ipm = ip_m_lookup(DL_OTHER);
4156                 ASSERT(ipm != NULL);
4157         }
4158         ill->ill_media = ipm;

4160         /*
4161          * When the new DLPI stuff is ready we'll pull lengths
4162          * from dlia.
4163          */
4164         if (dlia->dl_version == DL_VERSION_2) {
4165                 brdcst_addr_length = dlia->dl_brdcst_addr_length;
4166                 brdcst_addr = mi_offset_param(mp, dlia->dl_brdcst_addr_offset,
4167                     brdcst_addr_length);
4168                 if (brdcst_addr == NULL) {
4169                         brdcst_addr_length = 0;
4170                 }
4171                 sap_length = dlia->dl_sap_length;
4172                 phys_addr_length = dlia->dl_addr_length - ABS(sap_length);
4173                 ip1dbg(("ip: bcast_len %d, sap_len %d, phys_len %d\n",
4174                     brdcst_addr_length, sap_length, phys_addr_length));
4175         } else {
4176                 brdcst_addr_length = 6;
4177                 brdcst_addr = ip_six_byte_all_ones;
4178                 sap_length = -2;
4179                 phys_addr_length = brdcst_addr_length;
4180         }

4182         ill->ill_bcast_addr_length = brdcst_addr_length;
4183         ill->ill_phys_addr_length = phys_addr_length;
4184         ill->ill_sap_length = sap_length;

4186         /*
4187          * Synthetic DLPI types such as SUNW_DL_IPMP specify a zero SDU,
4188          * but we must ensure a minimum IP MTU is used since other bits of
4189          * IP will fly apart otherwise.
4190          */
4191         min_mtu = ill->ill_isv6 ? IPV6_MIN_MTU : IP_MIN_MTU;
4192         ill->ill_max_frag = MAX(min_mtu, dlia->dl_max_sdu);
4193         ill->ill_current_frag = ill->ill_max_frag;
4194         ill->ill_mtu = ill->ill_max_frag;
4195         ill->ill_mc_mtu = ill->ill_mtu; /* Overridden by DL_NOTE_SDU_SIZE2 */

4197         ill->ill_type = ipm->ip_m_type;

4199         if (!ill->ill_dlpi_style_set) {
4200                 if (dlia->dl_provider_style == DL_STYLE2)
4201                         ill->ill_needs_attach = 1;

4203                 phyint_flags_init(ill->ill_phyint, ill->ill_mactype);

4205                 /*
4206                  * Allocate the first ipif on this ill.  We don't delay it
4207                  * further as ioctl handling assumes at least one ipif exists.
4208                  *
4209                  * At this point we don't know whether the ill is v4 or v6.
4210                  * We will know this whan the SIOCSLIFNAME happens and
4211                  * the correct value for ill_isv6 will be assigned in
4212                  * ipif_set_values(). We need to hold the ill lock and
4213                  * clear the ILL_LL_SUBNET_PENDING flag and atomically do
4214                  * the wakeup.
4215                  */
4216                 (void) ipif_allocate(ill, 0, IRE_LOCAL,
4217                     dlia->dl_provider_style != DL_STYLE2, B_TRUE, NULL);
4218                 mutex_enter(&ill->ill_lock);
4219                 ASSERT(ill->ill_dlpi_style_set == 0);
```

```
4220                ill->ill_dlpi_style_set = 1;
4221                ill->ill_state_flags &= ~ILL_LL_SUBNET_PENDING;
4222                cv_broadcast(&ill->ill_cv);
4223                mutex_exit(&ill->ill_lock);
4224                freemsg(mp);
4225                return;
4226        }
4227        ASSERT(ill->ill_ipif != NULL);
4228        /*
4229         * We know whether it is IPv4 or IPv6 now, as this is the
4230         * second DL_INFO_ACK we are recieving in response to the
4231         * DL_INFO_REQ sent in ipif_set_values.
4232         */
4233        ill->ill_sap = (ill->ill_isv6) ? ipm->ip_m_ipv6sap : ipm->ip_m_ipv4sap;
4234        /*
4235         * Clear all the flags that were set based on ill_bcast_addr_length
4236         * and ill_phys_addr_length (in ipif_set_values) as these could have
4237         * changed now and we need to re-evaluate.
4238         */
4239        ill->ill_flags &= ~(ILLF_MULTICAST | ILLF_NONUD | ILLF_NOARP);
4240        ill->ill_ipif->ipif_flags &= ~(IPIF_BROADCAST | IPIF_POINTOPOINT);

4242        /*
4243         * Free ill_bcast_mp as things could have changed now.
4244         *
4245         * NOTE: The IPMP meta-interface is special-cased because it starts
4246         * with no underlying interfaces (and thus an unknown broadcast
4247         * address length), but we enforce that an interface is broadcast-
4248         * capable as part of allowing it to join a group.
4249         */
4250        if (ill->ill_bcast_addr_length == 0 && !IS_IPMP(ill)) {
4251                if (ill->ill_bcast_mp != NULL)
4252                        freemsg(ill->ill_bcast_mp);
4253                ill->ill_net_type = IRE_IF_NORESOLVER;

4255                ill->ill_bcast_mp = ill_dlur_gen(NULL,
4256                    ill->ill_phys_addr_length,
4257                    ill->ill_sap,
4258                    ill->ill_sap_length);

4260                if (ill->ill_isv6)
4261                        /*
4262                         * Note: xresolv interfaces will eventually need NOARP
4263                         * set here as well, but that will require those
4264                         * external resolvers to have some knowledge of
4265                         * that flag and act appropriately. Not to be changed
4266                         * at present.
4267                         */
4268                        ill->ill_flags |= ILLF_NONUD;
4269                else
4270                        ill->ill_flags |= ILLF_NOARP;

4272                if (ill->ill_mactype == SUNW_DL_VNI) {
4273                        ill->ill_ipif->ipif_flags |= IPIF_NOXMIT;
4274                } else if (ill->ill_phys_addr_length == 0 ||
4275                    ill->ill_mactype == DL_IPV4 ||
4276                    ill->ill_mactype == DL_IPV6) {
4277                        /*
4278                         * The underying link is point-to-point, so mark the
4279                         * interface as such.  We can do IP multicast over
4280                         * such a link since it transmits all network-layer
4281                         * packets to the remote side the same way.
4282                         */
4283                        ill->ill_flags |= ILLF_MULTICAST;
4284                        ill->ill_ipif->ipif_flags |= IPIF_POINTOPOINT;
4285                }
```



```
4286        } else {
4287                ill->ill_net_type = IRE_IF_RESOLVER;
4288                if (ill->ill_bcast_mp != NULL)
4289                        freemsg(ill->ill_bcast_mp);
4290                ill->ill_bcast_mp = ill_dlur_gen(brdcst_addr,
4291                    ill->ill_bcast_addr_length, ill->ill_sap,
4292                    ill->ill_sap_length);
4293                /*
4294                 * Later detect lack of DLPI driver multicast
4295                 * capability by catching DL_ENABMULTI errors in
4296                 * ip_rput_dlpi.
4297                 */
4298                ill->ill_flags |= ILLF_MULTICAST;
4299                if (!ill->ill_isv6)
4300                        ill->ill_ipif->ipif_flags |= IPIF_BROADCAST;
4301        }

4303        /* For IPMP, PHYI_IPMP should already be set by phyint_flags_init() */
4304        if (ill->ill_mactype == SUNW_DL_IPMP)
4305                ASSERT(ill->ill_phyint->phyint_flags & PHYI_IPMP);

4307        /* By default an interface does not support any CoS marking */
4308        ill->ill_flags &= ~ILLF_COS_ENABLED;

4310        /*
4311         * If we get QoS information in DL_INFO_ACK, the device supports
4312         * some form of CoS marking, set ILLF_COS_ENABLED.
4313         */
4314        sel1 = (dl_qos_cl_sel1_t *)mi_offset_param(mp, dlia->dl_qos_offset,
4315            dlia->dl_qos_length);
4316        if ((sel1 != NULL) && (sel1->dl_qos_type == DL_QOS_CL_SEL1)) {
4317                ill->ill_flags |= ILLF_COS_ENABLED;
4318        }

4320        /* Clear any previous error indication. */
4321        ill->ill_error = 0;
4322        freemsg(mp);
4323 }

4325 /*
4326  * Perform various checks to verify that an address would make sense as a
4327  * local, remote, or subnet interface address.
4328  */
4329 static boolean_t
4330 ip_addr_ok_v4(ipaddr_t addr, ipaddr_t subnet_mask)
4331 {
4332        ipaddr_t        net_mask;

4334        /*
4335         * Don't allow all zeroes, or all ones, but allow
4336         * all ones netmask.
4337         */
4338        if ((net_mask = ip_net_mask(addr)) == 0)
4339                return (B_FALSE);
4340        /* A given netmask overrides the "guess" netmask */
4341        if (subnet_mask != 0)
4342                net_mask = subnet_mask;
4343        if ((net_mask != ~(ipaddr_t)0) && ((addr == (addr & net_mask)) ||
4344            (addr == (addr | ~net_mask)))) {
4345                return (B_FALSE);
4346        }

4348        /*
4349         * Even if the netmask is all ones, we do not allow address to be
4350         * 255.255.255.255
4351         */
```

```
4352         if (addr == INADDR_BROADCAST)
4353                 return (B_FALSE);
4354
4355         if (CLASSD(addr))
4356                 return (B_FALSE);
4357
4358         return (B_TRUE);
4359 }
4360
4361 #define V6_IPIF_LINKLOCAL(p)    \
4362         IN6_IS_ADDR_LINKLOCAL(&(p)->ipif_v6lcl_addr)
4363
4364 /*
4365  * Compare two given ipifs and check if the second one is better than
4366  * the first one using the order of preference (not taking deprecated
4367  * into acount) specified in ipif_lookup_multicast().
4368  */
4369 static boolean_t
4370 ipif_comp_multi(ipif_t *old_ipif, ipif_t *new_ipif, boolean_t isv6)
4371 {
4372         /* Check the least preferred first. */
4373         if (IS_LOOPBACK(old_ipif->ipif_ill)) {
4374                 /* If both ipifs are the same, use the first one. */
4375                 if (IS_LOOPBACK(new_ipif->ipif_ill))
4376                         return (B_FALSE);
4377                 else
4378                         return (B_TRUE);
4379         }
4380
4381         /* For IPv6, check for link local address. */
4382         if (isv6 && V6_IPIF_LINKLOCAL(old_ipif)) {
4383                 if (IS_LOOPBACK(new_ipif->ipif_ill) ||
4384                     V6_IPIF_LINKLOCAL(new_ipif)) {
4385                         /* The second one is equal or less preferred. */
4386                         return (B_FALSE);
4387                 } else {
4388                         return (B_TRUE);
4389                 }
4390         }
4391
4392         /* Then check for point to point interface. */
4393         if (old_ipif->ipif_flags & IPIF_POINTOPOINT) {
4394                 if (IS_LOOPBACK(new_ipif->ipif_ill) ||
4395                     (isv6 && V6_IPIF_LINKLOCAL(new_ipif)) ||
4396                     (new_ipif->ipif_flags & IPIF_POINTOPOINT)) {
4397                         return (B_FALSE);
4398                 } else {
4399                         return (B_TRUE);
4400                 }
4401         }
4402
4403         /* old_ipif is a normal interface, so no need to use the new one. */
4404         return (B_FALSE);
4405 }
4406
4407 /*
4408  * Find a mulitcast-capable ipif given an IP instance and zoneid.
4409  * The ipif must be up, and its ill must multicast-capable, not
4410  * condemned, not an underlying interface in an IPMP group, and
4411  * not a VNI interface.  Order of preference:
4412  *
4413  *      1a. normal
4414  *      1b. normal, but deprecated
4415  *      2a. point to point
4416  *      2b. point to point, but deprecated
4417  *      3a. link local
```



```
4418  *      3b. link local, but deprecated
4419  *      4. loopback.
4420  */
4421 static ipif_t *
4422 ipif_lookup_multicast(ip_stack_t *ipst, zoneid_t zoneid, boolean_t isv6)
4423 {
4424         ill_t                   *ill;
4425         ill_walk_context_t      ctx;
4426         ipif_t                  *ipif;
4427         ipif_t                  *saved_ipif = NULL;
4428         ipif_t                  *dep_ipif = NULL;
4429
4430         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
4431         if (isv6)
4432                 ill = ILL_START_WALK_V6(&ctx, ipst);
4433         else
4434                 ill = ILL_START_WALK_V4(&ctx, ipst);
4435
4436         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
4437                 mutex_enter(&ill->ill_lock);
4438                 if (IS_VNI(ill) || IS_UNDER_IPMP(ill) ||
4439                     ILL_IS_CONDEMNED(ill) ||
4440                     !(ill->ill_flags & ILLF_MULTICAST)) {
4441                         mutex_exit(&ill->ill_lock);
4442                         continue;
4443                 }
4444                 for (ipif = ill->ill_ipif; ipif != NULL;
4445                     ipif = ipif->ipif_next) {
4446                         if (zoneid != ipif->ipif_zoneid &&
4447                             zoneid != ALL_ZONES &&
4448                             ipif->ipif_zoneid != ALL_ZONES) {
4449                                 continue;
4450                         }
4451                         if (!(ipif->ipif_flags & IPIF_UP) ||
4452                             IPIF_IS_CONDEMNED(ipif)) {
4453                                 continue;
4454                         }
4455
4456                         /*
4457                          * Found one candidate.  If it is deprecated,
4458                          * remember it in dep_ipif.  If it is not deprecated,
4459                          * remember it in saved_ipif.
4460                          */
4461                         if (ipif->ipif_flags & IPIF_DEPRECATED) {
4462                                 if (dep_ipif == NULL) {
4463                                         dep_ipif = ipif;
4464                                 } else if (ipif_comp_multi(dep_ipif, ipif,
4465                                     isv6)) {
4466                                         /*
4467                                          * If the previous dep_ipif does not
4468                                          * belong to the same ill, we've done
4469                                          * a ipif_refhold() on it.  So we need
4470                                          * to release it.
4471                                          */
4472                                         if (dep_ipif->ipif_ill != ill)
4473                                                 ipif_refrele(dep_ipif);
4474                                         dep_ipif = ipif;
4475                                 }
4476                                 continue;
4477                         }
4478                         if (saved_ipif == NULL) {
4479                                 saved_ipif = ipif;
4480                         } else {
4481                                 if (ipif_comp_multi(saved_ipif, ipif, isv6)) {
4482                                         if (saved_ipif->ipif_ill != ill)
4483                                                 ipif_refrele(saved_ipif);
```

```
4484                                        saved_ipif = ipif;
4485                                }
4486                        }
4487                }
4488                /*
4489                 * Before going to the next ill, do a ipif_refhold() on the
4490                 * saved ones.
4491                 */
4492                if (saved_ipif != NULL && saved_ipif->ipif_ill == ill)
4493                        ipif_refhold_locked(saved_ipif);
4494                if (dep_ipif != NULL && dep_ipif->ipif_ill == ill)
4495                        ipif_refhold_locked(dep_ipif);
4496                mutex_exit(&ill->ill_lock);
4497        }
4498        rw_exit(&ipst->ips_ill_g_lock);

4500        /*
4501         * If we have only the saved_ipif, return it.  But if we have both
4502         * saved_ipif and dep_ipif, check to see which one is better.
4503         */
4504        if (saved_ipif != NULL) {
4505                if (dep_ipif != NULL) {
4506                        if (ipif_comp_multi(saved_ipif, dep_ipif, isv6)) {
4507                                ipif_refrele(saved_ipif);
4508                                return (dep_ipif);
4509                        } else {
4510                                ipif_refrele(dep_ipif);
4511                                return (saved_ipif);
4512                        }
4513                }
4514                return (saved_ipif);
4515        } else {
4516                return (dep_ipif);
4517        }
4518 }

4520 ill_t *
4521 ill_lookup_multicast(ip_stack_t *ipst, zoneid_t zoneid, boolean_t isv6)
4522 {
4523        ipif_t *ipif;
4524        ill_t *ill;

4526        ipif = ipif_lookup_multicast(ipst, zoneid, isv6);
4527        if (ipif == NULL)
4528                return (NULL);

4530        ill = ipif->ipif_ill;
4531        ill_refhold(ill);
4532        ipif_refrele(ipif);
4533        return (ill);
4534 }

4536 /*
4537  * This function is called when an application does not specify an interface
4538  * to be used for multicast traffic (joining a group/sending data).  It
4539  * calls ire_lookup_multi() to look for an interface route for the
4540  * specified multicast group.  Doing this allows the administrator to add
4541  * prefix routes for multicast to indicate which interface to be used for
4542  * multicast traffic in the above scenario.  The route could be for all
4543  * multicast (224.0/4), for a single multicast group (a /32 route) or
4544  * anything in between.  If there is no such multicast route, we just find
4545  * any multicast capable interface and return it.  The returned ipif
4546  * is refhold'ed.
4547  *
4548  * We support MULTIRT and RTF_SETSRC on the multicast routes added to the
4549  * unicast table. This is used by CGTP.
```



```
4550  */
4551 ill_t *
4552 ill_lookup_group_v4(ipaddr_t group, zoneid_t zoneid, ip_stack_t *ipst,
4553     boolean_t *multirtp, ipaddr_t *setsrcp)
4554 {
4555        ill_t                   *ill;

4557        ill = ire_lookup_multi_ill_v4(group, zoneid, ipst, multirtp, setsrcp);
4558        if (ill != NULL)
4559                return (ill);

4561        return (ill_lookup_multicast(ipst, zoneid, B_FALSE));
4562 }

4564 /*
4565  * Look for an ipif with the specified interface address and destination.
4566  * The destination address is used only for matching point-to-point interfaces.
4567  */
4568 ipif_t *
4569 ipif_lookup_interface(ipaddr_t if_addr, ipaddr_t dst, ip_stack_t *ipst)
4570 {
4571        ipif_t  *ipif;
4572        ill_t   *ill;
4573        ill_walk_context_t ctx;

4575        /*
4576         * First match all the point-to-point interfaces
4577         * before looking at non-point-to-point interfaces.
4578         * This is done to avoid returning non-point-to-point
4579         * ipif instead of unnumbered point-to-point ipif.
4580         */
4581        rw_enter(&ipst->ips_ill_g_lock, RW_READER);
4582        ill = ILL_START_WALK_V4(&ctx, ipst);
4583        for (; ill != NULL; ill = ill_next(&ctx, ill)) {
4584                mutex_enter(&ill->ill_lock);
4585                for (ipif = ill->ill_ipif; ipif != NULL;
4586                    ipif = ipif->ipif_next) {
4587                        /* Allow the ipif to be down */
4588                        if ((ipif->ipif_flags & IPIF_POINTOPOINT) &&
4589                            (ipif->ipif_lcl_addr == if_addr) &&
4590                            (ipif->ipif_pp_dst_addr == dst)) {
4591                                if (!IPIF_IS_CONDEMNED(ipif)) {
4592                                        ipif_refhold_locked(ipif);
4593                                        mutex_exit(&ill->ill_lock);
4594                                        rw_exit(&ipst->ips_ill_g_lock);
4595                                        return (ipif);
4596                                }
4597                        }
4598                }
4599                mutex_exit(&ill->ill_lock);
4600        }
4601        rw_exit(&ipst->ips_ill_g_lock);

4603        /* lookup the ipif based on interface address */
4604        ipif = ipif_lookup_addr(if_addr, NULL, ALL_ZONES, ipst);
4605        ASSERT(ipif == NULL || !ipif->ipif_isv6);
4606        return (ipif);
4607 }

4609 /*
4610  * Common function for ipif_lookup_addr() and ipif_lookup_addr_exact().
4611  */
4612 static ipif_t *
4613 ipif_lookup_addr_common(ipaddr_t addr, ill_t *match_ill, uint32_t match_flags,
4614     zoneid_t zoneid, ip_stack_t *ipst)
4615 {
```

```
4616         ipif_t  *ipif;
4617         ill_t   *ill;
4618         boolean_t ptp = B_FALSE;
4619         ill_walk_context_t      ctx;
4620         boolean_t match_illgrp = (match_flags & IPIF_MATCH_ILLGRP);
4621         boolean_t no_duplicate = (match_flags & IPIF_MATCH_NONDUP);

4623         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
4624         /*
4625          * Repeat twice, first based on local addresses and
4626          * next time for pointopoint.
4627          */
4628 repeat:
4629         ill = ILL_START_WALK_V4(&ctx, ipst);
4630         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
4631                 if (match_ill != NULL && ill != match_ill &&
4632                     (!match_illgrp || !IS_IN_SAME_ILLGRP(ill, match_ill))) {
4633                         continue;
4634                 }
4635                 mutex_enter(&ill->ill_lock);
4636                 for (ipif = ill->ill_ipif; ipif != NULL;
4637                     ipif = ipif->ipif_next) {
4638                         if (zoneid != ALL_ZONES &&
4639                             zoneid != ipif->ipif_zoneid &&
4640                             ipif->ipif_zoneid != ALL_ZONES)
4641                                 continue;

4643                         if (no_duplicate && !(ipif->ipif_flags & IPIF_UP))
4644                                 continue;

4646                         /* Allow the ipif to be down */
4647                         if ((!ptp && (ipif->ipif_lcl_addr == addr) &&
4648                             ((ipif->ipif_flags & IPIF_UNNUMBERED) == 0)) ||
4649                             (ptp && (ipif->ipif_flags & IPIF_POINTOPOINT) &&
4650                             (ipif->ipif_pp_dst_addr == addr))) {
4651                                 if (!IPIF_IS_CONDEMNED(ipif)) {
4652                                         ipif_refhold_locked(ipif);
4653                                         mutex_exit(&ill->ill_lock);
4654                                         rw_exit(&ipst->ips_ill_g_lock);
4655                                         return (ipif);
4656                                 }
4657                         }
4658                 }
4659                 mutex_exit(&ill->ill_lock);
4660         }

4662         /* If we already did the ptp case, then we are done */
4663         if (ptp) {
4664                 rw_exit(&ipst->ips_ill_g_lock);
4665                 return (NULL);
4666         }
4667         ptp = B_TRUE;
4668         goto repeat;
4669 }

4671 /*
4672  * Lookup an ipif with the specified address.  For point-to-point links we
4673  * look for matches on either the destination address or the local address,
4674  * but we skip the local address check if IPIF_UNNUMBERED is set.  If the
4675  * 'match_ill' argument is non-NULL, the lookup is restricted to that ill
4676  * (or illgrp if 'match_ill' is in an IPMP group).
4677  */
4678 ipif_t *
4679 ipif_lookup_addr(ipaddr_t addr, ill_t *match_ill, zoneid_t zoneid,
4680     ip_stack_t *ipst)
4681 {
```



```
4682         return (ipif_lookup_addr_common(addr, match_ill, IPIF_MATCH_ILLGRP,
4683             zoneid, ipst));
4684 }

4686 /*
4687  * Lookup an ipif with the specified address. Similar to ipif_lookup_addr,
4688  * except that we will only return an address if it is not marked as
4689  * IPIF_DUPLICATE
4690  */
4691 ipif_t *
4692 ipif_lookup_addr_nondup(ipaddr_t addr, ill_t *match_ill, zoneid_t zoneid,
4693     ip_stack_t *ipst)
4694 {
4695         return (ipif_lookup_addr_common(addr, match_ill,
4696             (IPIF_MATCH_ILLGRP | IPIF_MATCH_NONDUP),
4697             zoneid, ipst));
4698 }

4700 /*
4701  * Special abbreviated version of ipif_lookup_addr() that doesn't match
4702  * 'match_ill' across the IPMP group.  This function is only needed in some
4703  * corner-cases; almost everything should use ipif_lookup_addr().
4704  */
4705 ipif_t *
4706 ipif_lookup_addr_exact(ipaddr_t addr, ill_t *match_ill, ip_stack_t *ipst)
4707 {
4708         ASSERT(match_ill != NULL);
4709         return (ipif_lookup_addr_common(addr, match_ill, 0, ALL_ZONES,
4710             ipst));
4711 }

4713 /*
4714  * Look for an ipif with the specified address. For point-point links
4715  * we look for matches on either the destination address and the local
4716  * address, but we ignore the check on the local address if IPIF_UNNUMBERED
4717  * is set.
4718  * If the 'match_ill' argument is non-NULL, the lookup is restricted to that
4719  * ill (or illgrp if 'match_ill' is in an IPMP group).
4720  * Return the zoneid for the ipif which matches. ALL_ZONES if no match.
4721  */
4722 zoneid_t
4723 ipif_lookup_addr_zoneid(ipaddr_t addr, ill_t *match_ill, ip_stack_t *ipst)
4724 {
4725         zoneid_t zoneid;
4726         ipif_t  *ipif;
4727         ill_t   *ill;
4728         boolean_t ptp = B_FALSE;
4729         ill_walk_context_t      ctx;

4731         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
4732         /*
4733          * Repeat twice, first based on local addresses and
4734          * next time for pointopoint.
4735          */
4736 repeat:
4737         ill = ILL_START_WALK_V4(&ctx, ipst);
4738         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
4739                 if (match_ill != NULL && ill != match_ill &&
4740                     !IS_IN_SAME_ILLGRP(ill, match_ill)) {
4741                         continue;
4742                 }
4743                 mutex_enter(&ill->ill_lock);
4744                 for (ipif = ill->ill_ipif; ipif != NULL;
4745                     ipif = ipif->ipif_next) {
4746                         /* Allow the ipif to be down */
4747                         if ((!ptp && (ipif->ipif_lcl_addr == addr) &&
```

```
4748                            ((ipif->ipif_flags & IPIF_UNNUMBERED) == 0)) ||
4749                            (ptp && (ipif->ipif_flags & IPIF_POINTOPOINT) &&
4750                            (ipif->ipif_pp_dst_addr == addr)) &&
4751                            !(ipif->ipif_state_flags & IPIF_CONDEMNED)) {
4752                                zoneid = ipif->ipif_zoneid;
4753                                mutex_exit(&ill->ill_lock);
4754                                rw_exit(&ipst->ips_ill_g_lock);
4755                                /*
4756                                 * If ipif_zoneid was ALL_ZONES then we have
4757                                 * a trusted extensions shared IP address.
4758                                 * In that case GLOBAL_ZONEID works to send.
4759                                 */
4760                                if (zoneid == ALL_ZONES)
4761                                        zoneid = GLOBAL_ZONEID;
4762                                return (zoneid);
4763                        }
4764                }
4765                mutex_exit(&ill->ill_lock);
4766        }

4768        /* If we already did the ptp case, then we are done */
4769        if (ptp) {
4770                rw_exit(&ipst->ips_ill_g_lock);
4771                return (ALL_ZONES);
4772        }
4773        ptp = B_TRUE;
4774        goto repeat;
4775 }

4777 /*
4778  * Look for an ipif that matches the specified remote address i.e. the
4779  * ipif that would receive the specified packet.
4780  * First look for directly connected interfaces and then do a recursive
4781  * IRE lookup and pick the first ipif corresponding to the source address in the
4782  * ire.
4783  * Returns: held ipif
4784  *
4785  * This is only used for ICMP_ADDRESS_MASK_REQUESTs
4786  */
4787 ipif_t *
4788 ipif_lookup_remote(ill_t *ill, ipaddr_t addr, zoneid_t zoneid)
4789 {
4790        ipif_t  *ipif;

4792        ASSERT(!ill->ill_isv6);

4794        /*
4795         * Someone could be changing this ipif currently or change it
4796         * after we return this. Thus  a few packets could use the old
4797         * old values. However structure updates/creates (ire, ilg, ilm etc)
4798         * will atomically be updated or cleaned up with the new value
4799         * Thus we don't need a lock to check the flags or other attrs below.
4800         */
4801        mutex_enter(&ill->ill_lock);
4802        for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
4803                if (IPIF_IS_CONDEMNED(ipif))
4804                        continue;
4805                if (zoneid != ALL_ZONES && zoneid != ipif->ipif_zoneid &&
4806                    ipif->ipif_zoneid != ALL_ZONES)
4807                        continue;
4808                /* Allow the ipif to be down */
4809                if (ipif->ipif_flags & IPIF_POINTOPOINT) {
4810                        if ((ipif->ipif_pp_dst_addr == addr) ||
4811                            (!(ipif->ipif_flags & IPIF_UNNUMBERED) &&
4812                            ipif->ipif_lcl_addr == addr)) {
4813                                ipif_refhold_locked(ipif);
```



```
4814                                mutex_exit(&ill->ill_lock);
4815                                return (ipif);
4816                        }
4817                } else if (ipif->ipif_subnet == (addr & ipif->ipif_net_mask)) {
4818                        ipif_refhold_locked(ipif);
4819                        mutex_exit(&ill->ill_lock);
4820                        return (ipif);
4821                }
4822        }
4823        mutex_exit(&ill->ill_lock);
4824        /*
4825         * For a remote destination it isn't possible to nail down a particular
4826         * ipif.
4827         */

4829        /* Pick the first interface */
4830        ipif = ipif_get_next_ipif(NULL, ill);
4831        return (ipif);
4832 }

4834 /*
4835  * This func does not prevent refcnt from increasing. But if
4836  * the caller has taken steps to that effect, then this func
4837  * can be used to determine whether the ill has become quiescent
4838  */
4839 static boolean_t
4840 ill_is_quiescent(ill_t *ill)
4841 {
4842        ipif_t  *ipif;

4844        ASSERT(MUTEX_HELD(&ill->ill_lock));

4846        for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
4847                if (ipif->ipif_refcnt != 0)
4848                        return (B_FALSE);
4849        }
4850        if (!ILL_DOWN_OK(ill) || ill->ill_refcnt != 0) {
4851                return (B_FALSE);
4852        }
4853        return (B_TRUE);
4854 }

4856 boolean_t
4857 ill_is_freeable(ill_t *ill)
4858 {
4859        ipif_t  *ipif;

4861        ASSERT(MUTEX_HELD(&ill->ill_lock));

4863        for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
4864                if (ipif->ipif_refcnt != 0) {
4865                        return (B_FALSE);
4866                }
4867        }
4868        if (!ILL_FREE_OK(ill) || ill->ill_refcnt != 0) {
4869                return (B_FALSE);
4870        }
4871        return (B_TRUE);
4872 }

4874 /*
4875  * This func does not prevent refcnt from increasing. But if
4876  * the caller has taken steps to that effect, then this func
4877  * can be used to determine whether the ipif has become quiescent
4878  */
4879 static boolean_t
```

```
4880 ipif_is_quiescent(ipif_t *ipif)
4881 {
4882         ill_t *ill;

4884         ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));

4886         if (ipif->ipif_refcnt != 0)
4887                 return (B_FALSE);

4889         ill = ipif->ipif_ill;
4890         if (ill->ill_ipif_up_count != 0 || ill->ill_ipif_dup_count != 0 ||
4891             ill->ill_logical_down) {
4892                 return (B_TRUE);
4893         }

4895         /* This is the last ipif going down or being deleted on this ill */
4896         if (ill->ill_ire_cnt != 0 || ill->ill_refcnt != 0) {
4897                 return (B_FALSE);
4898         }

4900         return (B_TRUE);
4901 }

4903 /*
4904  * return true if the ipif can be destroyed: the ipif has to be quiescent
4905  * with zero references from ire/ilm to it.
4906  */
4907 static boolean_t
4908 ipif_is_freeable(ipif_t *ipif)
4909 {
4910         ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));
4911         ASSERT(ipif->ipif_id != 0);
4912         return (ipif->ipif_refcnt == 0);
4913 }

4915 /*
4916  * The ipif/ill/ire has been refreled. Do the tail processing.
4917  * Determine if the ipif or ill in question has become quiescent and if so
4918  * wakeup close and/or restart any queued pending ioctl that is waiting
4919  * for the ipif_down (or ill_down)
4920  */
4921 void
4922 ipif_ill_refrele_tail(ill_t *ill)
4923 {
4924         mblk_t  *mp;
4925         conn_t  *connp;
4926         ipsq_t  *ipsq;
4927         ipxop_t *ipx;
4928         ipif_t  *ipif;
4929         dl_notify_ind_t *dlindp;

4931         ASSERT(MUTEX_HELD(&ill->ill_lock));

4933         if ((ill->ill_state_flags & ILL_CONDEMNED) && ill_is_freeable(ill)) {
4934                 /* ip_modclose() may be waiting */
4935                 cv_broadcast(&ill->ill_cv);
4936         }

4938         ipsq = ill->ill_phyint->phyint_ipsq;
4939         mutex_enter(&ipsq->ipsq_lock);
4940         ipx = ipsq->ipsq_xop;
4941         mutex_enter(&ipx->ipx_lock);
4942         if (ipx->ipx_waitfor == 0)      /* no one's waiting; bail */
4943                 goto unlock;

4945         ASSERT(ipx->ipx_pending_mp != NULL && ipx->ipx_pending_ipif != NULL);
```



```
4947         ipif = ipx->ipx_pending_ipif;
4948         if (ipif->ipif_ill != ill)      /* wait is for another ill; bail */
4949                 goto unlock;

4951         switch (ipx->ipx_waitfor) {
4952         case IPIF_DOWN:
4953                 if (!ipif_is_quiescent(ipif))
4954                         goto unlock;
4955                 break;
4956         case IPIF_FREE:
4957                 if (!ipif_is_freeable(ipif))
4958                         goto unlock;
4959                 break;
4960         case ILL_DOWN:
4961                 if (!ill_is_quiescent(ill))
4962                         goto unlock;
4963                 break;
4964         case ILL_FREE:
4965                 /*
4966                  * ILL_FREE is only for loopback; normal ill teardown waits
4967                  * synchronously in ip_modclose() without using ipx_waitfor,
4968                  * handled by the cv_broadcast() at the top of this function.
4969                  */
4970                 if (!ill_is_freeable(ill))
4971                         goto unlock;
4972                 break;
4973         default:
4974                 cmn_err(CE_PANIC, "ipsq: %p unknown ipx_waitfor %d\n",
4975                     (void *)ipsq, ipx->ipx_waitfor);
4976         }

4978         ill_refhold_locked(ill);        /* for qwriter_ip() call below */
4979         mutex_exit(&ipx->ipx_lock);
4980         mp = ipsq_pending_mp_get(ipsq, &connp);
4981         mutex_exit(&ipsq->ipsq_lock);
4982         mutex_exit(&ill->ill_lock);

4984         ASSERT(mp != NULL);
4985         /*
4986          * NOTE: all of the qwriter_ip() calls below use CUR_OP since
4987          * we can only get here when the current operation decides it
4988          * it needs to quiesce via ipsq_pending_mp_add().
4989          */
4990         switch (mp->b_datap->db_type) {
4991         case M_PCPROTO:
4992         case M_PROTO:
4993                 /*
4994                  * For now, only DL_NOTIFY_IND messages can use this facility.
4995                  */
4996                 dlindp = (dl_notify_ind_t *)mp->b_rptr;
4997                 ASSERT(dlindp->dl_primitive == DL_NOTIFY_IND);

4999                 switch (dlindp->dl_notification) {
5000                 case DL_NOTE_PHYS_ADDR:
5001                         qwriter_ip(ill, ill->ill_rq, mp,
5002                             ill_set_phys_addr_tail, CUR_OP, B_TRUE);
5003                         return;
5004                 case DL_NOTE_REPLUMB:
5005                         qwriter_ip(ill, ill->ill_rq, mp,
5006                             ill_replumb_tail, CUR_OP, B_TRUE);
5007                         return;
5008                 default:
5009                         ASSERT(0);
5010                         ill_refrele(ill);
5011                 }
```

```
5012                    break;

5014            case M_ERROR:
5015            case M_HANGUP:
5016                    qwriter_ip(ill, ill->ill_rq, mp, ipif_all_down_tail, CUR_OP,
5017                        B_TRUE);
5018                    return;

5020            case M_IOCTL:
5021            case M_IOCDATA:
5022                    qwriter_ip(ill, (connp != NULL ? CONNP_TO_WQ(connp) :
5023                        ill->ill_wq), mp, ip_reprocess_ioctl, CUR_OP, B_TRUE);
5024                    return;

5026            default:
5027                    cmn_err(CE_PANIC, "ipif_ill_refrele_tail mp %p "
5028                        "db_type %d\n", (void *)mp, mp->b_datap->db_type);
5029            }
5030            return;
5031    unlock:
5032            mutex_exit(&ipsq->ipsq_lock);
5033            mutex_exit(&ipx->ipx_lock);
5034            mutex_exit(&ill->ill_lock);
5035    }

5037    #ifdef DEBUG
5038    /* Reuse trace buffer from beginning (if reached the end) and record trace */
5039    static void
5040    th_trace_rrecord(th_trace_t *th_trace)
5041    {
5042            tr_buf_t *tr_buf;
5043            uint_t lastref;

5045            lastref = th_trace->th_trace_lastref;
5046            lastref++;
5047            if (lastref == TR_BUF_MAX)
5048                    lastref = 0;
5049            th_trace->th_trace_lastref = lastref;
5050            tr_buf = &th_trace->th_trbuf[lastref];
5051            tr_buf->tr_time = ddi_get_lbolt();
5052            tr_buf->tr_depth = getpcstack(tr_buf->tr_stack, TR_STACK_DEPTH);
5053    }

5055    static void
5056    th_trace_free(void *value)
5057    {
5058            th_trace_t *th_trace = value;

5060            ASSERT(th_trace->th_refcnt == 0);
5061            kmem_free(th_trace, sizeof (*th_trace));
5062    }

5064    /*
5065     * Find or create the per-thread hash table used to track object references.
5066     * The ipst argument is NULL if we shouldn't allocate.
5067     *
5068     * Accesses per-thread data, so there's no need to lock here.
5069     */
5070    static mod_hash_t *
5071    th_trace_gethash(ip_stack_t *ipst)
5072    {
5073            th_hash_t *thh;

5075            if ((thh = tsd_get(ip_thread_data)) == NULL && ipst != NULL) {
5076                    mod_hash_t *mh;
5077                    char name[256];
```



```
5078                    size_t objsize, rshift;
5079                    int retv;

5081                    if ((thh = kmem_alloc(sizeof (*thh), KM_NOSLEEP)) == NULL)
5082                            return (NULL);
5083                    (void) snprintf(name, sizeof (name), "th_trace_%p",
5084                        (void *)curthread);

5086                    /*
5087                     * We use mod_hash_create_extended here rather than the more
5088                     * obvious mod_hash_create_ptrhash because the latter has a
5089                     * hard-coded KM_SLEEP, and we'd prefer to fail rather than
5090                     * block.
5091                     */
5092                    objsize = MAX(MAX(sizeof (ill_t), sizeof (ipif_t)),
5093                        MAX(sizeof (ire_t), sizeof (ncec_t)));
5094                    rshift = highbit(objsize);
5095                    mh = mod_hash_create_extended(name, 64, mod_hash_null_keydtor,
5096                        th_trace_free, mod_hash_byptr, (void *)rshift,
5097                        mod_hash_ptrkey_cmp, KM_NOSLEEP);
5098                    if (mh == NULL) {
5099                            kmem_free(thh, sizeof (*thh));
5100                            return (NULL);
5101                    }
5102                    thh->thh_hash = mh;
5103                    thh->thh_ipst = ipst;
5104                    /*
5105                     * We trace ills, ipifs, ires, and nces.  All of these are
5106                     * per-IP-stack, so the lock on the thread list is as well.
5107                     */
5108                    rw_enter(&ip_thread_rwlock, RW_WRITER);
5109                    list_insert_tail(&ip_thread_list, thh);
5110                    rw_exit(&ip_thread_rwlock);
5111                    retv = tsd_set(ip_thread_data, thh);
5112                    ASSERT(retv == 0);
5113            }
5114            return (thh != NULL ? thh->thh_hash : NULL);
5115    }

5117    boolean_t
5118    th_trace_ref(const void *obj, ip_stack_t *ipst)
5119    {
5120            th_trace_t *th_trace;
5121            mod_hash_t *mh;
5122            mod_hash_val_t val;

5124            if ((mh = th_trace_gethash(ipst)) == NULL)
5125                    return (B_FALSE);

5127            /*
5128             * Attempt to locate the trace buffer for this obj and thread.
5129             * If it does not exist, then allocate a new trace buffer and
5130             * insert into the hash.
5131             */
5132            if (mod_hash_find(mh, (mod_hash_key_t)obj, &val) == MH_ERR_NOTFOUND) {
5133                    th_trace = kmem_zalloc(sizeof (th_trace_t), KM_NOSLEEP);
5134                    if (th_trace == NULL)
5135                            return (B_FALSE);

5137                    th_trace->th_id = curthread;
5138                    if (mod_hash_insert(mh, (mod_hash_key_t)obj,
5139                        (mod_hash_val_t)th_trace) != 0) {
5140                            kmem_free(th_trace, sizeof (th_trace_t));
5141                            return (B_FALSE);
5142                    }
5143            } else {
```

```
5144                 th_trace = (th_trace_t *)val;
5145         }

5147         ASSERT(th_trace->th_refcnt >= 0 &&
5148             th_trace->th_refcnt < TR_BUF_MAX - 1);

5150         th_trace->th_refcnt++;
5151         th_trace_rrecord(th_trace);
5152         return (B_TRUE);
5153 }

5155 /*
5156  * For the purpose of tracing a reference release, we assume that global
5157  * tracing is always on and that the same thread initiated the reference hold
5158  * is releasing.
5159  */
5160 void
5161 th_trace_unref(const void *obj)
5162 {
5163         int retv;
5164         mod_hash_t *mh;
5165         th_trace_t *th_trace;
5166         mod_hash_val_t val;

5168         mh = th_trace_gethash(NULL);
5169         retv = mod_hash_find(mh, (mod_hash_key_t)obj, &val);
5170         ASSERT(retv == 0);
5171         th_trace = (th_trace_t *)val;

5173         ASSERT(th_trace->th_refcnt > 0);
5174         th_trace->th_refcnt--;
5175         th_trace_rrecord(th_trace);
5176 }

5178 /*
5179  * If tracing has been disabled, then we assume that the reference counts are
5180  * now useless, and we clear them out before destroying the entries.
5181  */
5182 void
5183 th_trace_cleanup(const void *obj, boolean_t trace_disable)
5184 {
5185         th_hash_t       *thh;
5186         mod_hash_t      *mh;
5187         mod_hash_val_t  val;
5188         th_trace_t      *th_trace;
5189         int             retv;

5191         rw_enter(&ip_thread_rwlock, RW_READER);
5192         for (thh = list_head(&ip_thread_list); thh != NULL;
5193             thh = list_next(&ip_thread_list, thh)) {
5194                 if (mod_hash_find(mh = thh->thh_hash, (mod_hash_key_t)obj,
5195                     &val) == 0) {
5196                         th_trace = (th_trace_t *)val;
5197                         if (trace_disable)
5198                                 th_trace->th_refcnt = 0;
5199                         retv = mod_hash_destroy(mh, (mod_hash_key_t)obj);
5200                         ASSERT(retv == 0);
5201                 }
5202         }
5203         rw_exit(&ip_thread_rwlock);
5204 }

5206 void
5207 ipif_trace_ref(ipif_t *ipif)
5208 {
5209         ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));
```



```
5211         if (ipif->ipif_trace_disable)
5212                 return;

5214         if (!th_trace_ref(ipif, ipif->ipif_ill->ill_ipst)) {
5215                 ipif->ipif_trace_disable = B_TRUE;
5216                 ipif_trace_cleanup(ipif);
5217         }
5218 }

5220 void
5221 ipif_untrace_ref(ipif_t *ipif)
5222 {
5223         ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));

5225         if (!ipif->ipif_trace_disable)
5226                 th_trace_unref(ipif);
5227 }

5229 void
5230 ill_trace_ref(ill_t *ill)
5231 {
5232         ASSERT(MUTEX_HELD(&ill->ill_lock));

5234         if (ill->ill_trace_disable)
5235                 return;

5237         if (!th_trace_ref(ill, ill->ill_ipst)) {
5238                 ill->ill_trace_disable = B_TRUE;
5239                 ill_trace_cleanup(ill);
5240         }
5241 }

5243 void
5244 ill_untrace_ref(ill_t *ill)
5245 {
5246         ASSERT(MUTEX_HELD(&ill->ill_lock));

5248         if (!ill->ill_trace_disable)
5249                 th_trace_unref(ill);
5250 }

5252 /*
5253  * Called when ipif is unplumbed or when memory alloc fails.  Note that on
5254  * failure, ipif_trace_disable is set.
5255  */
5256 static void
5257 ipif_trace_cleanup(const ipif_t *ipif)
5258 {
5259         th_trace_cleanup(ipif, ipif->ipif_trace_disable);
5260 }

5262 /*
5263  * Called when ill is unplumbed or when memory alloc fails.  Note that on
5264  * failure, ill_trace_disable is set.
5265  */
5266 static void
5267 ill_trace_cleanup(const ill_t *ill)
5268 {
5269         th_trace_cleanup(ill, ill->ill_trace_disable);
5270 }
5271 #endif /* DEBUG */

5273 void
5274 ipif_refhold_locked(ipif_t *ipif)
5275 {
```

```
5276         ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));
5277         ipif->ipif_refcnt++;
5278         IPIF_TRACE_REF(ipif);
5279 }

5281 void
5282 ipif_refhold(ipif_t *ipif)
5283 {
5284         ill_t   *ill;

5286         ill = ipif->ipif_ill;
5287         mutex_enter(&ill->ill_lock);
5288         ipif->ipif_refcnt++;
5289         IPIF_TRACE_REF(ipif);
5290         mutex_exit(&ill->ill_lock);
5291 }

5293 /*
5294  * Must not be called while holding any locks. Otherwise if this is
5295  * the last reference to be released there is a chance of recursive mutex
5296  * panic due to ipif_refrele -> ipif_ill_refrele_tail -> qwriter_ip trying
5297  * to restart an ioctl.
5298  */
5299 void
5300 ipif_refrele(ipif_t *ipif)
5301 {
5302         ill_t   *ill;

5304         ill = ipif->ipif_ill;

5306         mutex_enter(&ill->ill_lock);
5307         ASSERT(ipif->ipif_refcnt != 0);
5308         ipif->ipif_refcnt--;
5309         IPIF_UNTRACE_REF(ipif);
5310         if (ipif->ipif_refcnt != 0) {
5311                 mutex_exit(&ill->ill_lock);
5312                 return;
5313         }

5315         /* Drops the ill_lock */
5316         ipif_ill_refrele_tail(ill);
5317 }

5319 ipif_t *
5320 ipif_get_next_ipif(ipif_t *curr, ill_t *ill)
5321 {
5322         ipif_t  *ipif;

5324         mutex_enter(&ill->ill_lock);
5325         for (ipif = (curr == NULL ? ill->ill_ipif : curr->ipif_next);
5326             ipif != NULL; ipif = ipif->ipif_next) {
5327                 if (IPIF_IS_CONDEMNED(ipif))
5328                         continue;
5329                 ipif_refhold_locked(ipif);
5330                 mutex_exit(&ill->ill_lock);
5331                 return (ipif);
5332         }
5333         mutex_exit(&ill->ill_lock);
5334         return (NULL);
5335 }

5337 /*
5338  * TODO: make this table extendible at run time
5339  * Return a pointer to the mac type info for 'mac_type'
5340  */
5341 static ip_m_t *
```



```
5342 ip_m_lookup(t_uscalar_t mac_type)
5343 {
5344         ip_m_t  *ipm;

5346         for (ipm = ip_m_tbl; ipm < A_END(ip_m_tbl); ipm++)
5347                 if (ipm->ip_m_mac_type == mac_type)
5348                         return (ipm);
5349         return (NULL);
5350 }

5352 /*
5353  * Make a link layer address from the multicast IP address *addr.
5354  * To form the link layer address, invoke the ip_m_v*mapping function
5355  * associated with the link-layer type.
5356  */
5357 void
5358 ip_mcast_mapping(ill_t *ill, uchar_t *addr, uchar_t *hwaddr)
5359 {
5360         ip_m_t *ipm;

5362         if (ill->ill_net_type == IRE_IF_NORESOLVER)
5363                 return;

5365         ASSERT(addr != NULL);

5367         ipm = ip_m_lookup(ill->ill_mactype);
5368         if (ipm == NULL ||
5369             (ill->ill_isv6 && ipm->ip_m_v6mapping == NULL) ||
5370             (!ill->ill_isv6 && ipm->ip_m_v4mapping == NULL)) {
5371                 ip0dbg(("no mapping for ill %s mactype 0x%x\n",
5372                     ill->ill_name, ill->ill_mactype));
5373                 return;
5374         }
5375         if (ill->ill_isv6)
5376                 (*ipm->ip_m_v6mapping)(ill, addr, hwaddr);
5377         else
5378                 (*ipm->ip_m_v4mapping)(ill, addr, hwaddr);
5379 }

5381 /*
5382  * Returns B_FALSE if the IPv4 netmask pointed by 'mask' is non-contiguous.
5383  * Otherwise returns B_TRUE.
5384  *
5385  * The netmask can be verified to be contiguous with 32 shifts and or
5386  * operations. Take the contiguous mask (in host byte order) and compute
5387  *      mask | mask << 1 | mask << 2 | ... | mask << 31
5388  * the result will be the same as the 'mask' for contiguous mask.
5389  */
5390 static boolean_t
5391 ip_contiguous_mask(uint32_t mask)
5392 {
5393         uint32_t        m = mask;
5394         int             i;

5396         for (i = 1; i < 32; i++)
5397                 m |= (mask << i);

5399         return (m == mask);
5400 }

5402 /*
5403  * ip_rt_add is called to add an IPv4 route to the forwarding table.
5404  * ill is passed in to associate it with the correct interface.
5405  * If ire_arg is set, then we return the held IRE in that location.
5406  */
5407 int
```

```
5408 ip_rt_add(ipaddr_t dst_addr, ipaddr_t mask, ipaddr_t gw_addr,
5409     ipaddr_t src_addr, int flags, ill_t *ill, ire_t **ire_arg,
5410     boolean_t ioctl_msg, struct rtsa_s *sp, ip_stack_t *ipst, zoneid_t zoneid)
5411 {
5412         ire_t   *ire, *nire;
5413         ire_t   *gw_ire = NULL;
5414         ipif_t  *ipif = NULL;
5415         uint_t  type;
5416         int     match_flags = MATCH_IRE_TYPE;
5417         tsol_gc_t *gc = NULL;
5418         tsol_gcgrp_t *gcgrp = NULL;
5419         boolean_t gcgrp_xtraref = B_FALSE;
5420         boolean_t cgtp_broadcast;
5421         boolean_t unbound = B_FALSE;

5423         ip1dbg(("ip_rt_add:"));

5425         if (ire_arg != NULL)
5426                 *ire_arg = NULL;

5428         /* disallow non-contiguous netmasks */
5429         if (!ip_contiguous_mask(ntohl(mask)))
5430                 return (ENOTSUP);

5432         /*
5433          * If this is the case of RTF_HOST being set, then we set the netmask
5434          * to all ones (regardless if one was supplied).
5435          */
5436         if (flags & RTF_HOST)
5437                 mask = IP_HOST_MASK;

5439         /*
5440          * Prevent routes with a zero gateway from being created (since
5441          * interfaces can currently be plumbed and brought up no assigned
5442          * address).
5443          */
5444         if (gw_addr == 0)
5445                 return (ENETUNREACH);
5446         /*
5447          * Get the ipif, if any, corresponding to the gw_addr
5448          * If -ifp was specified we restrict ourselves to the ill, otherwise
5449          * we match on the gatway and destination to handle unnumbered pt-pt
5450          * interfaces.
5451          */
5452         if (ill != NULL)
5453                 ipif = ipif_lookup_addr(gw_addr, ill, ALL_ZONES, ipst);
5454         else
5455                 ipif = ipif_lookup_interface(gw_addr, dst_addr, ipst);
5456         if (ipif != NULL) {
5457                 if (IS_VNI(ipif->ipif_ill)) {
5458                         ipif_refrele(ipif);
5459                         return (EINVAL);
5460                 }
5461         }

5463         /*
5464          * GateD will attempt to create routes with a loopback interface
5465          * address as the gateway and with RTF_GATEWAY set.  We allow
5466          * these routes to be added, but create them as interface routes
5467          * since the gateway is an interface address.
5468          */
5469         if ((ipif != NULL) && (ipif->ipif_ire_type == IRE_LOOPBACK)) {
5470                 flags &= ~RTF_GATEWAY;
5471                 if (gw_addr == INADDR_LOOPBACK && dst_addr == INADDR_LOOPBACK &&
5472                     mask == IP_HOST_MASK) {
5473                         ire = ire_ftable_lookup_v4(dst_addr, 0, 0, IRE_LOOPBACK,
```



```
5474                             NULL, ALL_ZONES, NULL, MATCH_IRE_TYPE, 0, ipst,
5475                             NULL);
5476                         if (ire != NULL) {
5477                                 ire_refrele(ire);
5478                                 ipif_refrele(ipif);
5479                                 return (EEXIST);
5480                         }
5481                         ip1dbg(("ip_rt_add: 0x%p creating IRE 0x%x"
5482                             "for 0x%x\n", (void *)ipif,
5483                             ipif->ipif_ire_type,
5484                             ntohl(ipif->ipif_lcl_addr)));
5485                         ire = ire_create(
5486                             (uchar_t *)&dst_addr,       /* dest address */
5487                             (uchar_t *)&mask,           /* mask */
5488                             NULL,                       /* no gateway */
5489                             ipif->ipif_ire_type,        /* LOOPBACK */
5490                             ipif->ipif_ill,
5491                             zoneid,
5492                             (ipif->ipif_flags & IPIF_PRIVATE) ? RTF_PRIVATE : 0,
5493                             NULL,
5494                             ipst);

5496                         if (ire == NULL) {
5497                                 ipif_refrele(ipif);
5498                                 return (ENOMEM);
5499                         }
5500                         /* src address assigned by the caller? */
5501                         if ((src_addr != INADDR_ANY) && (flags & RTF_SETSRC))
5502                                 ire->ire_setsrc_addr = src_addr;

5504                         nire = ire_add(ire);
5505                         if (nire == NULL) {
5506                                 /*
5507                                  * In the result of failure, ire_add() will have
5508                                  * already deleted the ire in question, so there
5509                                  * is no need to do that here.
5510                                  */
5511                                 ipif_refrele(ipif);
5512                                 return (ENOMEM);
5513                         }
5514                         /*
5515                          * Check if it was a duplicate entry. This handles
5516                          * the case of two racing route adds for the same route
5517                          */
5518                         if (nire != ire) {
5519                                 ASSERT(nire->ire_identical_ref > 1);
5520                                 ire_delete(nire);
5521                                 ire_refrele(nire);
5522                                 ipif_refrele(ipif);
5523                                 return (EEXIST);
5524                         }
5525                         ire = nire;
5526                         goto save_ire;
5527                 }
5528         }

5530         /*
5531          * The routes for multicast with CGTP are quite special in that
5532          * the gateway is the local interface address, yet RTF_GATEWAY
5533          * is set. We turn off RTF_GATEWAY to provide compatibility with
5534          * this undocumented and unusual use of multicast routes.
5535          */
5536         if ((flags & RTF_MULTIRT) && ipif != NULL)
5537                 flags &= ~RTF_GATEWAY;

5539         /*
```

```
5540          * Traditionally, interface routes are ones where RTF_GATEWAY isn't set
5541          * and the gateway address provided is one of the system's interface
5542          * addresses.  By using the routing socket interface and supplying an
5543          * RTA_IFP sockaddr with an interface index, an alternate method of
5544          * specifying an interface route to be created is available which uses
5545          * the interface index that specifies the outgoing interface rather than
5546          * the address of an outgoing interface (which may not be able to
5547          * uniquely identify an interface).  When coupled with the RTF_GATEWAY
5548          * flag, routes can be specified which not only specify the next-hop to
5549          * be used when routing to a certain prefix, but also which outgoing
5550          * interface should be used.
5551          *
5552          * Previously, interfaces would have unique addresses assigned to them
5553          * and so the address assigned to a particular interface could be used
5554          * to identify a particular interface.  One exception to this was the
5555          * case of an unnumbered interface (where IPIF_UNNUMBERED was set).
5556          *
5557          * With the advent of IPv6 and its link-local addresses, this
5558          * restriction was relaxed and interfaces could share addresses between
5559          * themselves.  In fact, typically all of the link-local interfaces on
5560          * an IPv6 node or router will have the same link-local address.  In
5561          * order to differentiate between these interfaces, the use of an
5562          * interface index is necessary and this index can be carried inside a
5563          * RTA_IFP sockaddr (which is actually a sockaddr_dl).  One restriction
5564          * of using the interface index, however, is that all of the ipif's that
5565          * are part of an ill have the same index and so the RTA_IFP sockaddr
5566          * cannot be used to differentiate between ipif's (or logical
5567          * interfaces) that belong to the same ill (physical interface).
5568          *
5569          * For example, in the following case involving IPv4 interfaces and
5570          * logical interfaces
5571          *
5572          *      192.0.2.32      255.255.255.224 192.0.2.33      U       if0
5573          *      192.0.2.32      255.255.255.224 192.0.2.34      U       if0
5574          *      192.0.2.32      255.255.255.224 192.0.2.35      U       if0
5575          *
5576          * the ipif's corresponding to each of these interface routes can be
5577          * uniquely identified by the "gateway" (actually interface address).
5578          *
5579          * In this case involving multiple IPv6 default routes to a particular
5580          * link-local gateway, the use of RTA_IFP is necessary to specify which
5581          * default route is of interest:
5582          *
5583          *      default         fe80::123:4567:89ab:cdef        U       if0
5584          *      default         fe80::123:4567:89ab:cdef        U       if1
5585          */

5587         /* RTF_GATEWAY not set */
5588         if (!(flags & RTF_GATEWAY)) {
5589                 if (sp != NULL) {
5590                         ip2dbg(("ip_rt_add: gateway security attributes "
5591                             "cannot be set with interface route\n"));
5592                         if (ipif != NULL)
5593                                 ipif_refrele(ipif);
5594                         return (EINVAL);
5595                 }

5597                 /*
5598                  * Whether or not ill (RTA_IFP) is set, we require that
5599                  * the gateway is one of our local addresses.
5600                  */
5601                 if (ipif == NULL)
5602                         return (ENETUNREACH);

5604                 /*
5605                  * We use MATCH_IRE_ILL here. If the caller specified an
```



```
5606                  * interface (from the RTA_IFP sockaddr) we use it, otherwise
5607                  * we use the ill derived from the gateway address.
5608                  * We can always match the gateway address since we record it
5609                  * in ire_gateway_addr.
5610                  * We don't allow RTA_IFP to specify a different ill than the
5611                  * one matching the ipif to make sure we can delete the route.
5612                  */
5613                 match_flags |= MATCH_IRE_GW | MATCH_IRE_ILL;
5614                 if (ill == NULL) {
5615                         ill = ipif->ipif_ill;
5616                 } else if (ill != ipif->ipif_ill) {
5617                         ipif_refrele(ipif);
5618                         return (EINVAL);
5619                 }

5621                 /*
5622                  * We check for an existing entry at this point.
5623                  *
5624                  * Since a netmask isn't passed in via the ioctl interface
5625                  * (SIOCADDRT), we don't check for a matching netmask in that
5626                  * case.
5627                  */
5628                 if (!ioctl_msg)
5629                         match_flags |= MATCH_IRE_MASK;
5630                 ire = ire_ftable_lookup_v4(dst_addr, mask, gw_addr,
5631                     IRE_INTERFACE, ill, ALL_ZONES, NULL, match_flags, 0, ipst,
5632                     NULL);
5633                 if (ire != NULL) {
5634                         ire_refrele(ire);
5635                         ipif_refrele(ipif);
5636                         return (EEXIST);
5637                 }

5639                 /*
5640                  * Some software (for example, GateD and Sun Cluster) attempts
5641                  * to create (what amount to) IRE_PREFIX routes with the
5642                  * loopback address as the gateway.  This is primarily done to
5643                  * set up prefixes with the RTF_REJECT flag set (for example,
5644                  * when generating aggregate routes.)
5645                  *
5646                  * If the IRE type (as defined by ill->ill_net_type) would be
5647                  * IRE_LOOPBACK, then we map the request into a
5648                  * IRE_IF_NORESOLVER. We also OR in the RTF_BLACKHOLE flag as
5649                  * these interface routes, by definition, can only be that.
5650                  *
5651                  * Needless to say, the real IRE_LOOPBACK is NOT created by this
5652                  * routine, but rather using ire_create() directly.
5653                  *
5654                  */
5655                 type = ill->ill_net_type;
5656                 if (type == IRE_LOOPBACK) {
5657                         type = IRE_IF_NORESOLVER;
5658                         flags |= RTF_BLACKHOLE;
5659                 }

5661                 /*
5662                  * Create a copy of the IRE_IF_NORESOLVER or
5663                  * IRE_IF_RESOLVER with the modified address, netmask, and
5664                  * gateway.
5665                  */
5666                 ire = ire_create(
5667                     (uchar_t *)&dst_addr,
5668                     (uint8_t *)&mask,
5669                     (uint8_t *)&gw_addr,
5670                     type,
5671                     ill,
```

```
5672                    zoneid,
5673                    flags,
5674                    NULL,
5675                    ipst);
5676                if (ire == NULL) {
5677                        ipif_refrele(ipif);
5678                        return (ENOMEM);
5679                }

5681                /* src address assigned by the caller? */
5682                if ((src_addr != INADDR_ANY) && (flags & RTF_SETSRC))
5683                        ire->ire_setsrc_addr = src_addr;

5685                nire = ire_add(ire);
5686                if (nire == NULL) {
5687                        /*
5688                         * In the result of failure, ire_add() will have
5689                         * already deleted the ire in question, so there
5690                         * is no need to do that here.
5691                         */
5692                        ipif_refrele(ipif);
5693                        return (ENOMEM);
5694                }
5695                /*
5696                 * Check if it was a duplicate entry. This handles
5697                 * the case of two racing route adds for the same route
5698                 */
5699                if (nire != ire) {
5700                        ire_delete(nire);
5701                        ire_refrele(nire);
5702                        ipif_refrele(ipif);
5703                        return (EEXIST);
5704                }
5705                ire = nire;
5706                goto save_ire;
5707        }

5709        /*
5710         * Get an interface IRE for the specified gateway.
5711         * If we don't have an IRE_IF_NORESOLVER or IRE_IF_RESOLVER for the
5712         * gateway, it is currently unreachable and we fail the request
5713         * accordingly. We reject any RTF_GATEWAY routes where the gateway
5714         * is an IRE_LOCAL or IRE_LOOPBACK.
5715         * If RTA_IFP was specified we look on that particular ill.
5716         */
5717        if (ill != NULL)
5718                match_flags |= MATCH_IRE_ILL;

5720        /* Check whether the gateway is reachable. */
5721 again:
5722        type = IRE_INTERFACE | IRE_LOCAL | IRE_LOOPBACK;
5723        if (flags & RTF_INDIRECT)
5724                type |= IRE_OFFLINK;

5726        gw_ire = ire_ftable_lookup_v4(gw_addr, 0, 0, type, ill,
5727            ALL_ZONES, NULL, match_flags, 0, ipst, NULL);
5728        if (gw_ire == NULL) {
5729                /*
5730                 * With IPMP, we allow host routes to influence in.mpathd's
5731                 * target selection.  However, if the test addresses are on
5732                 * their own network, the above lookup will fail since the
5733                 * underlying IRE_INTERFACEs are marked hidden.  So allow
5734                 * hidden test IREs to be found and try again.
5735                 */
5736                if (!(match_flags & MATCH_IRE_TESTHIDDEN))  {
5737                        match_flags |= MATCH_IRE_TESTHIDDEN;
```



```
5738                        goto again;
5739                }
5740                if (ipif != NULL)
5741                        ipif_refrele(ipif);
5742                return (ENETUNREACH);
5743        }
5744        if (gw_ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK)) {
5745                ire_refrele(gw_ire);
5746                if (ipif != NULL)
5747                        ipif_refrele(ipif);
5748                return (ENETUNREACH);
5749        }

5751        if (ill == NULL && !(flags & RTF_INDIRECT)) {
5752                unbound = B_TRUE;
5753                if (ipst->ips_ip_strict_src_multihoming > 0)
5754                        ill = gw_ire->ire_ill;
5755        }

5757        /*
5758         * We create one of three types of IREs as a result of this request
5759         * based on the netmask.  A netmask of all ones (which is automatically
5760         * assumed when RTF_HOST is set) results in an IRE_HOST being created.
5761         * An all zeroes netmask implies a default route so an IRE_DEFAULT is
5762         * created.  Otherwise, an IRE_PREFIX route is created for the
5763         * destination prefix.
5764         */
5765        if (mask == IP_HOST_MASK)
5766                type = IRE_HOST;
5767        else if (mask == 0)
5768                type = IRE_DEFAULT;
5769        else
5770                type = IRE_PREFIX;

5772        /* check for a duplicate entry */
5773        ire = ire_ftable_lookup_v4(dst_addr, mask, gw_addr, type, ill,
5774            ALL_ZONES, NULL, match_flags | MATCH_IRE_MASK | MATCH_IRE_GW,
5775            0, ipst, NULL);
5776        if (ire != NULL) {
5777                if (ipif != NULL)
5778                        ipif_refrele(ipif);
5779                ire_refrele(gw_ire);
5780                ire_refrele(ire);
5781                return (EEXIST);
5782        }

5784        /* Security attribute exists */
5785        if (sp != NULL) {
5786                tsol_gcgrp_addr_t ga;

5788                /* find or create the gateway credentials group */
5789                ga.ga_af = AF_INET;
5790                IN6_IPADDR_TO_V4MAPPED(gw_addr, &ga.ga_addr);

5792                /* we hold reference to it upon success */
5793                gcgrp = gcgrp_lookup(&ga, B_TRUE);
5794                if (gcgrp == NULL) {
5795                        if (ipif != NULL)
5796                                ipif_refrele(ipif);
5797                        ire_refrele(gw_ire);
5798                        return (ENOMEM);
5799                }

5801                /*
5802                 * Create and add the security attribute to the group; a
5803                 * reference to the group is made upon allocating a new
```

```
5804                 * entry successfully.  If it finds an already-existing
5805                 * entry for the security attribute in the group, it simply
5806                 * returns it and no new reference is made to the group.
5807                 */
5808                gc = gc_create(sp, gcgrp, &gcgrp_xtraref);
5809                if (gc == NULL) {
5810                        if (ipif != NULL)
5811                                ipif_refrele(ipif);
5812                        /* release reference held by gcgrp_lookup */
5813                        GCGRP_REFRELE(gcgrp);
5814                        ire_refrele(gw_ire);
5815                        return (ENOMEM);
5816                }
5817        }

5819        /* Create the IRE. */
5820        ire = ire_create(
5821            (uchar_t *)&dst_addr,               /* dest address */
5822            (uchar_t *)&mask,                   /* mask */
5823            (uchar_t *)&gw_addr,                /* gateway address */
5824            (ushort_t)type,                     /* IRE type */
5825            ill,
5826            zoneid,
5827            flags,
5828            gc,                                 /* security attribute */
5829            ipst);

5831        /*
5832         * The ire holds a reference to the 'gc' and the 'gc' holds a
5833         * reference to the 'gcgrp'. We can now release the extra reference
5834         * the 'gcgrp' acquired in the gcgrp_lookup, if it was not used.
5835         */
5836        if (gcgrp_xtraref)
5837                GCGRP_REFRELE(gcgrp);
5838        if (ire == NULL) {
5839                if (gc != NULL)
5840                        GC_REFRELE(gc);
5841                if (ipif != NULL)
5842                        ipif_refrele(ipif);
5843                ire_refrele(gw_ire);
5844                return (ENOMEM);
5845        }

5847        /* Before we add, check if an extra CGTP broadcast is needed */
5848        cgtp_broadcast = ((flags & RTF_MULTIRT) &&
5849            ip_type_v4(ire->ire_addr, ipst) == IRE_BROADCAST);

5851        /* src address assigned by the caller? */
5852        if ((src_addr != INADDR_ANY) && (flags & RTF_SETSRC))
5853                ire->ire_setsrc_addr = src_addr;

5855        ire->ire_unbound = unbound;

5857        /*
5858         * POLICY: should we allow an RTF_HOST with address INADDR_ANY?
5859         * SUN/OS socket stuff does but do we really want to allow 0.0.0.0?
5860         */

5862        /* Add the new IRE. */
5863        nire = ire_add(ire);
5864        if (nire == NULL) {
5865                /*
5866                 * In the result of failure, ire_add() will have
5867                 * already deleted the ire in question, so there
5868                 * is no need to do that here.
5869                 */
```



```
5870                if (ipif != NULL)
5871                        ipif_refrele(ipif);
5872                ire_refrele(gw_ire);
5873                return (ENOMEM);
5874        }
5875        /*
5876         * Check if it was a duplicate entry. This handles
5877         * the case of two racing route adds for the same route
5878         */
5879        if (nire != ire) {
5880                ire_delete(nire);
5881                ire_refrele(nire);
5882                if (ipif != NULL)
5883                        ipif_refrele(ipif);
5884                ire_refrele(gw_ire);
5885                return (EEXIST);
5886        }
5887        ire = nire;

5889        if (flags & RTF_MULTIRT) {
5890                /*
5891                 * Invoke the CGTP (multirouting) filtering module
5892                 * to add the dst address in the filtering database.
5893                 * Replicated inbound packets coming from that address
5894                 * will be filtered to discard the duplicates.
5895                 * It is not necessary to call the CGTP filter hook
5896                 * when the dst address is a broadcast or multicast,
5897                 * because an IP source address cannot be a broadcast
5898                 * or a multicast.
5899                 */
5900                if (cgtp_broadcast) {
5901                        ip_cgtp_bcast_add(ire, ipst);
5902                        goto save_ire;
5903                }
5904                if (ipst->ips_ip_cgtp_filter_ops != NULL &&
5905                    !CLASSD(ire->ire_addr)) {
5906                        int res;
5907                        ipif_t *src_ipif;

5909                        /* Find the source address corresponding to gw_ire */
5910                        src_ipif = ipif_lookup_addr(gw_ire->ire_gateway_addr,
5911                            NULL, zoneid, ipst);
5912                        if (src_ipif != NULL) {
5913                                res = ipst->ips_ip_cgtp_filter_ops->
5914                                    cfo_add_dest_v4(
5915                                    ipst->ips_netstack->netstack_stackid,
5916                                    ire->ire_addr,
5917                                    ire->ire_gateway_addr,
5918                                    ire->ire_setsrc_addr,
5919                                    src_ipif->ipif_lcl_addr);
5920                                ipif_refrele(src_ipif);
5921                        } else {
5922                                res = EADDRNOTAVAIL;
5923                        }
5924                        if (res != 0) {
5925                                if (ipif != NULL)
5926                                        ipif_refrele(ipif);
5927                                ire_refrele(gw_ire);
5928                                ire_delete(ire);
5929                                ire_refrele(ire);       /* Held in ire_add */
5930                                return (res);
5931                        }
5932                }
5933        }

5935 save_ire:
```

```
5936         if (gw_ire != NULL) {
5937                 ire_refrele(gw_ire);
5938                 gw_ire = NULL;
5939         }
5940         if (ill != NULL) {
5941                 /*
5942                  * Save enough information so that we can recreate the IRE if
5943                  * the interface goes down and then up.  The metrics associated
5944                  * with the route will be saved as well when rts_setmetrics() is
5945                  * called after the IRE has been created.  In the case where
5946                  * memory cannot be allocated, none of this information will be
5947                  * saved.
5948                  */
5949                 ill_save_ire(ill, ire);
5950         }
5951         if (ioctl_msg)
5952                 ip_rts_rtmsg(RTM_OLDADD, ire, 0, ipst);
5953         if (ire_arg != NULL) {
5954                 /*
5955                  * Store the ire that was successfully added into where ire_arg
5956                  * points to so that callers don't have to look it up
5957                  * themselves (but they are responsible for ire_refrele()ing
5958                  * the ire when they are finished with it).
5959                  */
5960                 *ire_arg = ire;
5961         } else {
5962                 ire_refrele(ire);               /* Held in ire_add */
5963         }
5964         if (ipif != NULL)
5965                 ipif_refrele(ipif);
5966         return (0);
5967 }

5969 /*
5970  * ip_rt_delete is called to delete an IPv4 route.
5971  * ill is passed in to associate it with the correct interface.
5972  */
5973 /* ARGSUSED4 */
5974 int
5975 ip_rt_delete(ipaddr_t dst_addr, ipaddr_t mask, ipaddr_t gw_addr,
5976     uint_t rtm_addrs, int flags, ill_t *ill, boolean_t ioctl_msg,
5977     ip_stack_t *ipst, zoneid_t zoneid)
5978 {
5979         ire_t   *ire = NULL;
5980         ipif_t  *ipif;
5981         uint_t  type;
5982         uint_t  match_flags = MATCH_IRE_TYPE;
5983         int     err = 0;

5985         ip1dbg(("ip_rt_delete:"));
5986         /*
5987          * If this is the case of RTF_HOST being set, then we set the netmask
5988          * to all ones.  Otherwise, we use the netmask if one was supplied.
5989          */
5990         if (flags & RTF_HOST) {
5991                 mask = IP_HOST_MASK;
5992                 match_flags |= MATCH_IRE_MASK;
5993         } else if (rtm_addrs & RTA_NETMASK) {
5994                 match_flags |= MATCH_IRE_MASK;
5995         }

5997         /*
5998          * Note that RTF_GATEWAY is never set on a delete, therefore
5999          * we check if the gateway address is one of our interfaces first,
6000          * and fall back on RTF_GATEWAY routes.
6001          *
```



```
6002          * This makes it possible to delete an original
6003          * IRE_IF_NORESOLVER/IRE_IF_RESOLVER - consistent with SunOS 4.1.
6004          * However, we have RTF_KERNEL set on the ones created by ipif_up
6005          * and those can not be deleted here.
6006          *
6007          * We use MATCH_IRE_ILL if we know the interface. If the caller
6008          * specified an interface (from the RTA_IFP sockaddr) we use it,
6009          * otherwise we use the ill derived from the gateway address.
6010          * We can always match the gateway address since we record it
6011          * in ire_gateway_addr.
6012          *
6013          * For more detail on specifying routes by gateway address and by
6014          * interface index, see the comments in ip_rt_add().
6015          */
6016         ipif = ipif_lookup_interface(gw_addr, dst_addr, ipst);
6017         if (ipif != NULL) {
6018                 ill_t   *ill_match;

6020                 if (ill != NULL)
6021                         ill_match = ill;
6022                 else
6023                         ill_match = ipif->ipif_ill;

6025                 match_flags |= MATCH_IRE_ILL;
6026                 if (ipif->ipif_ire_type == IRE_LOOPBACK) {
6027                         ire = ire_ftable_lookup_v4(dst_addr, mask, 0,
6028                             IRE_LOOPBACK, ill_match, ALL_ZONES, NULL,
6029                             match_flags, 0, ipst, NULL);
6030                 }
6031                 if (ire == NULL) {
6032                         match_flags |= MATCH_IRE_GW;
6033                         ire = ire_ftable_lookup_v4(dst_addr, mask, gw_addr,
6034                             IRE_INTERFACE, ill_match, ALL_ZONES, NULL,
6035                             match_flags, 0, ipst, NULL);
6036                 }
6037                 /* Avoid deleting routes created by kernel from an ipif */
6038                 if (ire != NULL && (ire->ire_flags & RTF_KERNEL)) {
6039                         ire_refrele(ire);
6040                         ire = NULL;
6041                 }

6043                 /* Restore in case we didn't find a match */
6044                 match_flags &= ~(MATCH_IRE_GW|MATCH_IRE_ILL);
6045         }

6047         if (ire == NULL) {
6048                 /*
6049                  * At this point, the gateway address is not one of our own
6050                  * addresses or a matching interface route was not found.  We
6051                  * set the IRE type to lookup based on whether
6052                  * this is a host route, a default route or just a prefix.
6053                  *
6054                  * If an ill was passed in, then the lookup is based on an
6055                  * interface index so MATCH_IRE_ILL is added to match_flags.
6056                  */
6057                 match_flags |= MATCH_IRE_GW;
6058                 if (ill != NULL)
6059                         match_flags |= MATCH_IRE_ILL;
6060                 if (mask == IP_HOST_MASK)
6061                         type = IRE_HOST;
6062                 else if (mask == 0)
6063                         type = IRE_DEFAULT;
6064                 else
6065                         type = IRE_PREFIX;
6066                 ire = ire_ftable_lookup_v4(dst_addr, mask, gw_addr, type, ill,
6067                     ALL_ZONES, NULL, match_flags, 0, ipst, NULL);
```

```
6068         }

6070         if (ipif != NULL) {
6071                 ipif_refrele(ipif);
6072                 ipif = NULL;
6073         }

6075         if (ire == NULL)
6076                 return (ESRCH);

6078         if (ire->ire_flags & RTF_MULTIRT) {
6079                 /*
6080                  * Invoke the CGTP (multirouting) filtering module
6081                  * to remove the dst address from the filtering database.
6082                  * Packets coming from that address will no longer be
6083                  * filtered to remove duplicates.
6084                  */
6085                 if (ipst->ips_ip_cgtp_filter_ops != NULL) {
6086                         err = ipst->ips_ip_cgtp_filter_ops->cfo_del_dest_v4(
6087                             ipst->ips_netstack->netstack_stackid,
6088                             ire->ire_addr, ire->ire_gateway_addr);
6089                 }
6090                 ip_cgtp_bcast_delete(ire, ipst);
6091         }

6093         ill = ire->ire_ill;
6094         if (ill != NULL)
6095                 ill_remove_saved_ire(ill, ire);
6096         if (ioctl_msg)
6097                 ip_rts_rtmsg(RTM_OLDDEL, ire, 0, ipst);
6098         ire_delete(ire);
6099         ire_refrele(ire);
6100         return (err);
6101 }

6103 /*
6104  * ip_siocaddrt is called to complete processing of an SIOCADDRT IOCTL.
6105  */
6106 /* ARGSUSED */
6107 int
6108 ip_siocaddrt(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
6109     ip_ioctl_cmd_t *ipip, void *dummy_if_req)
6110 {
6111         ipaddr_t dst_addr;
6112         ipaddr_t gw_addr;
6113         ipaddr_t mask;
6114         int error = 0;
6115         mblk_t *mp1;
6116         struct rtentry *rt;
6117         ipif_t *ipif = NULL;
6118         ip_stack_t      *ipst;

6120         ASSERT(q->q_next == NULL);
6121         ipst = CONNQ_TO_IPST(q);

6123         ip1dbg(("ip_siocaddrt:"));
6124         /* Existence of mp1 verified in ip_wput_nondata */
6125         mp1 = mp->b_cont->b_cont;
6126         rt = (struct rtentry *)mp1->b_rptr;

6128         dst_addr = ((sin_t *)&rt->rt_dst)->sin_addr.s_addr;
6129         gw_addr = ((sin_t *)&rt->rt_gateway)->sin_addr.s_addr;

6131         /*
6132          * If the RTF_HOST flag is on, this is a request to assign a gateway
6133          * to a particular host address.  In this case, we set the netmask to
```



```
6134          * all ones for the particular destination address.  Otherwise,
6135          * determine the netmask to be used based on dst_addr and the interfaces
6136          * in use.
6137          */
6138         if (rt->rt_flags & RTF_HOST) {
6139                 mask = IP_HOST_MASK;
6140         } else {
6141                 /*
6142                  * Note that ip_subnet_mask returns a zero mask in the case of
6143                  * default (an all-zeroes address).
6144                  */
6145                 mask = ip_subnet_mask(dst_addr, &ipif, ipst);
6146         }

6148         error = ip_rt_add(dst_addr, mask, gw_addr, 0, rt->rt_flags, NULL, NULL,
6149             B_TRUE, NULL, ipst, ALL_ZONES);
6150         if (ipif != NULL)
6151                 ipif_refrele(ipif);
6152         return (error);
6153 }

6155 /*
6156  * ip_siocdelrt is called to complete processing of an SIOCDELRT IOCTL.
6157  */
6158 /* ARGSUSED */
6159 int
6160 ip_siocdelrt(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
6161     ip_ioctl_cmd_t *ipip, void *dummy_if_req)
6162 {
6163         ipaddr_t dst_addr;
6164         ipaddr_t gw_addr;
6165         ipaddr_t mask;
6166         int error;
6167         mblk_t *mp1;
6168         struct rtentry *rt;
6169         ipif_t *ipif = NULL;
6170         ip_stack_t      *ipst;

6172         ASSERT(q->q_next == NULL);
6173         ipst = CONNQ_TO_IPST(q);

6175         ip1dbg(("ip_siocdelrt:"));
6176         /* Existence of mp1 verified in ip_wput_nondata */
6177         mp1 = mp->b_cont->b_cont;
6178         rt = (struct rtentry *)mp1->b_rptr;

6180         dst_addr = ((sin_t *)&rt->rt_dst)->sin_addr.s_addr;
6181         gw_addr = ((sin_t *)&rt->rt_gateway)->sin_addr.s_addr;

6183         /*
6184          * If the RTF_HOST flag is on, this is a request to delete a gateway
6185          * to a particular host address.  In this case, we set the netmask to
6186          * all ones for the particular destination address.  Otherwise,
6187          * determine the netmask to be used based on dst_addr and the interfaces
6188          * in use.
6189          */
6190         if (rt->rt_flags & RTF_HOST) {
6191                 mask = IP_HOST_MASK;
6192         } else {
6193                 /*
6194                  * Note that ip_subnet_mask returns a zero mask in the case of
6195                  * default (an all-zeroes address).
6196                  */
6197                 mask = ip_subnet_mask(dst_addr, &ipif, ipst);
6198         }
```

```
6200         error = ip_rt_delete(dst_addr, mask, gw_addr,
6201             RTA_DST | RTA_GATEWAY | RTA_NETMASK, rt->rt_flags, NULL, B_TRUE,
6202             ipst, ALL_ZONES);
6203         if (ipif != NULL)
6204                 ipif_refrele(ipif);
6205         return (error);
6206 }

6208 /*
6209  * Enqueue the mp onto the ipsq, chained by b_next.
6210  * b_prev stores the function to be executed later, and b_queue the queue
6211  * where this mp originated.
6212  */
6213 void
6214 ipsq_enq(ipsq_t *ipsq, queue_t *q, mblk_t *mp, ipsq_func_t func, int type,
6215     ill_t *pending_ill)
6216 {
6217         conn_t  *connp;
6218         ipxop_t *ipx = ipsq->ipsq_xop;

6220         ASSERT(MUTEX_HELD(&ipsq->ipsq_lock));
6221         ASSERT(MUTEX_HELD(&ipx->ipx_lock));
6222         ASSERT(func != NULL);

6224         mp->b_queue = q;
6225         mp->b_prev = (void *)func;
6226         mp->b_next = NULL;

6228         switch (type) {
6229         case CUR_OP:
6230                 if (ipx->ipx_mptail != NULL) {
6231                         ASSERT(ipx->ipx_mphead != NULL);
6232                         ipx->ipx_mptail->b_next = mp;
6233                 } else {
6234                         ASSERT(ipx->ipx_mphead == NULL);
6235                         ipx->ipx_mphead = mp;
6236                 }
6237                 ipx->ipx_mptail = mp;
6238                 break;

6240         case NEW_OP:
6241                 if (ipsq->ipsq_xopq_mptail != NULL) {
6242                         ASSERT(ipsq->ipsq_xopq_mphead != NULL);
6243                         ipsq->ipsq_xopq_mptail->b_next = mp;
6244                 } else {
6245                         ASSERT(ipsq->ipsq_xopq_mphead == NULL);
6246                         ipsq->ipsq_xopq_mphead = mp;
6247                 }
6248                 ipsq->ipsq_xopq_mptail = mp;
6249                 ipx->ipx_ipsq_queued = B_TRUE;
6250                 break;

6252         case SWITCH_OP:
6253                 ASSERT(ipsq->ipsq_swxop != NULL);
6254                 /* only one switch operation is currently allowed */
6255                 ASSERT(ipsq->ipsq_switch_mp == NULL);
6256                 ipsq->ipsq_switch_mp = mp;
6257                 ipx->ipx_ipsq_queued = B_TRUE;
6258                 break;
6259         default:
6260                 cmn_err(CE_PANIC, "ipsq_enq %d type \n", type);
6261         }

6263         if (CONN_Q(q) && pending_ill != NULL) {
6264                 connp = Q_TO_CONN(q);
6265                 ASSERT(MUTEX_HELD(&connp->conn_lock));
```



```
6266                 connp->conn_oper_pending_ill = pending_ill;
6267         }
6268 }

6270 /*
6271  * Dequeue the next message that requested exclusive access to this IPSQ's
6272  * xop.  Specifically:
6273  *
6274  *  1. If we're still processing the current operation on 'ipsq', then
6275  *     dequeue the next message for the operation (from ipx_mphead), or
6276  *     return NULL if there are no queued messages for the operation.
6277  *     These messages are queued via CUR_OP to qwriter_ip() and friends.
6278  *
6279  *  2. If the current operation on 'ipsq' has completed (ipx_current_ipif is
6280  *     not set) see if the ipsq has requested an xop switch.  If so, switch
6281  *     'ipsq' to a different xop.  Xop switches only happen when joining or
6282  *     leaving IPMP groups and require a careful dance -- see the comments
6283  *     in-line below for details.  If we're leaving a group xop or if we're
6284  *     joining a group xop and become writer on it, then we proceed to (3).
6285  *     Otherwise, we return NULL and exit the xop.
6286  *
6287  *  3. For each IPSQ in the xop, return any switch operation stored on
6288  *     ipsq_switch_mp (set via SWITCH_OP); these must be processed before
6289  *     any other messages queued on the IPSQ.  Otherwise, dequeue the next
6290  *     exclusive operation (queued via NEW_OP) stored on ipsq_xopq_mphead.
6291  *     Note that if the phyint tied to 'ipsq' is not using IPMP there will
6292  *     only be one IPSQ in the xop.  Otherwise, there will be one IPSQ for
6293  *     each phyint in the group, including the IPMP meta-interface phyint.
6294  */
6295 static mblk_t *
6296 ipsq_dq(ipsq_t *ipsq)
6297 {
6298         ill_t   *illv4, *illv6;
6299         mblk_t  *mp;
6300         ipsq_t  *xopipsq;
6301         ipsq_t  *leftipsq = NULL;
6302         ipxop_t *ipx;
6303         phyint_t *phyi = ipsq->ipsq_phyint;
6304         ip_stack_t *ipst = ipsq->ipsq_ipst;
6305         boolean_t emptied = B_FALSE;

6307         /*
6308          * Grab all the locks we need in the defined order (ill_g_lock ->
6309          * ipsq_lock -> ipx_lock); ill_g_lock is needed to use ipsq_next.
6310          */
6311         rw_enter(&ipst->ips_ill_g_lock,
6312             ipsq->ipsq_swxop != NULL ? RW_WRITER : RW_READER);
6313         mutex_enter(&ipsq->ipsq_lock);
6314         ipx = ipsq->ipsq_xop;
6315         mutex_enter(&ipx->ipx_lock);

6317         /*
6318          * Dequeue the next message associated with the current exclusive
6319          * operation, if any.
6320          */
6321         if ((mp = ipx->ipx_mphead) != NULL) {
6322                 ipx->ipx_mphead = mp->b_next;
6323                 if (ipx->ipx_mphead == NULL)
6324                         ipx->ipx_mptail = NULL;
6325                 mp->b_next = (void *)ipsq;
6326                 goto out;
6327         }

6329         if (ipx->ipx_current_ipif != NULL)
6330                 goto empty;
```

```
6332         if (ipsq->ipsq_swxop != NULL) {
6333                 /*
6334                  * The exclusive operation that is now being completed has
6335                  * requested a switch to a different xop.  This happens
6336                  * when an interface joins or leaves an IPMP group.  Joins
6337                  * happen through SIOCSLIFGROUPNAME (ip_sioctl_groupname()).
6338                  * Leaves happen via SIOCSLIFGROUPNAME, interface unplumb
6339                  * (phyint_free()), or interface plumb for an ill type
6340                  * not in the IPMP group (ip_rput_dlpi_writer()).
6341                  *
6342                  * Xop switches are not allowed on the IPMP meta-interface.
6343                  */
6344                 ASSERT(phyi == NULL || !(phyi->phyint_flags & PHYI_IPMP));
6345                 ASSERT(RW_WRITE_HELD(&ipst->ips_ill_g_lock));
6346                 DTRACE_PROBE1(ipsq__switch, (ipsq_t *), ipsq);

6348                 if (ipsq->ipsq_swxop == &ipsq->ipsq_ownxop) {
6349                         /*
6350                          * We're switching back to our own xop, so we have two
6351                          * xop's to drain/exit: our own, and the group xop
6352                          * that we are leaving.
6353                          *
6354                          * First, pull ourselves out of the group ipsq list.
6355                          * This is safe since we're writer on ill_g_lock.
6356                          */
6357                         ASSERT(ipsq->ipsq_xop != &ipsq->ipsq_ownxop);

6359                         xopipsq = ipx->ipx_ipsq;
6360                         while (xopipsq->ipsq_next != ipsq)
6361                                 xopipsq = xopipsq->ipsq_next;

6363                         xopipsq->ipsq_next = ipsq->ipsq_next;
6364                         ipsq->ipsq_next = ipsq;
6365                         ipsq->ipsq_xop = ipsq->ipsq_swxop;
6366                         ipsq->ipsq_swxop = NULL;

6368                         /*
6369                          * Second, prepare to exit the group xop.  The actual
6370                          * ipsq_exit() is done at the end of this function
6371                          * since we cannot hold any locks across ipsq_exit().
6372                          * Note that although we drop the group's ipx_lock, no
6373                          * threads can proceed since we're still ipx_writer.
6374                          */
6375                         leftipsq = xopipsq;
6376                         mutex_exit(&ipx->ipx_lock);

6378                         /*
6379                          * Third, set ipx to point to our own xop (which was
6380                          * inactive and therefore can be entered).
6381                          */
6382                         ipx = ipsq->ipsq_xop;
6383                         mutex_enter(&ipx->ipx_lock);
6384                         ASSERT(ipx->ipx_writer == NULL);
6385                         ASSERT(ipx->ipx_current_ipif == NULL);
6386                 } else {
6387                         /*
6388                          * We're switching from our own xop to a group xop.
6389                          * The requestor of the switch must ensure that the
6390                          * group xop cannot go away (e.g. by ensuring the
6391                          * phyint associated with the xop cannot go away).
6392                          *
6393                          * If we can become writer on our new xop, then we'll
6394                          * do the drain.  Otherwise, the current writer of our
6395                          * new xop will do the drain when it exits.
6396                          *
6397                          * First, splice ourselves into the group IPSQ list.
```



```
6398                          * This is safe since we're writer on ill_g_lock.
6399                          */
6400                         ASSERT(ipsq->ipsq_xop == &ipsq->ipsq_ownxop);

6402                         xopipsq = ipsq->ipsq_swxop->ipx_ipsq;
6403                         while (xopipsq->ipsq_next != ipsq->ipsq_swxop->ipx_ipsq)
6404                                 xopipsq = xopipsq->ipsq_next;

6406                         xopipsq->ipsq_next = ipsq;
6407                         ipsq->ipsq_next = ipsq->ipsq_swxop->ipx_ipsq;
6408                         ipsq->ipsq_xop = ipsq->ipsq_swxop;
6409                         ipsq->ipsq_swxop = NULL;

6411                         /*
6412                          * Second, exit our own xop, since it's now unused.
6413                          * This is safe since we've got the only reference.
6414                          */
6415                         ASSERT(ipx->ipx_writer == curthread);
6416                         ipx->ipx_writer = NULL;
6417                         VERIFY(--ipx->ipx_reentry_cnt == 0);
6418                         ipx->ipx_ipsq_queued = B_FALSE;
6419                         mutex_exit(&ipx->ipx_lock);

6421                         /*
6422                          * Third, set ipx to point to our new xop, and check
6423                          * if we can become writer on it.  If we cannot, then
6424                          * the current writer will drain the IPSQ group when
6425                          * it exits.  Our ipsq_xop is guaranteed to be stable
6426                          * because we're still holding ipsq_lock.
6427                          */
6428                         ipx = ipsq->ipsq_xop;
6429                         mutex_enter(&ipx->ipx_lock);
6430                         if (ipx->ipx_writer != NULL ||
6431                             ipx->ipx_current_ipif != NULL) {
6432                                 goto out;
6433                         }
6434                 }

6436                 /*
6437                  * Fourth, become writer on our new ipx before we continue
6438                  * with the drain.  Note that we never dropped ipsq_lock
6439                  * above, so no other thread could've raced with us to
6440                  * become writer first.  Also, we're holding ipx_lock, so
6441                  * no other thread can examine the ipx right now.
6442                  */
6443                 ASSERT(ipx->ipx_current_ipif == NULL);
6444                 ASSERT(ipx->ipx_mphead == NULL && ipx->ipx_mptail == NULL);
6445                 VERIFY(ipx->ipx_reentry_cnt++ == 0);
6446                 ipx->ipx_writer = curthread;
6447                 ipx->ipx_forced = B_FALSE;
6448 #ifdef DEBUG
6449                 ipx->ipx_depth = getpcstack(ipx->ipx_stack, IPX_STACK_DEPTH);
6450 #endif
6451         }

6453         xopipsq = ipsq;
6454         do {
6455                 /*
6456                  * So that other operations operate on a consistent and
6457                  * complete phyint, a switch message on an IPSQ must be
6458                  * handled prior to any other operations on that IPSQ.
6459                  */
6460                 if ((mp = xopipsq->ipsq_switch_mp) != NULL) {
6461                         xopipsq->ipsq_switch_mp = NULL;
6462                         ASSERT(mp->b_next == NULL);
6463                         mp->b_next = (void *)xopipsq;
```

```
6464                        goto out;
6465                }

6467                if ((mp = xopipsq->ipsq_xopq_mphead) != NULL) {
6468                        xopipsq->ipsq_xopq_mphead = mp->b_next;
6469                        if (xopipsq->ipsq_xopq_mphead == NULL)
6470                                xopipsq->ipsq_xopq_mptail = NULL;
6471                        mp->b_next = (void *)xopipsq;
6472                        goto out;
6473                }
6474        } while ((xopipsq = xopipsq->ipsq_next) != ipsq);
6475 empty:
6476        /*
6477         * There are no messages.  Further, we are holding ipx_lock, hence no
6478         * new messages can end up on any IPSQ in the xop.
6479         */
6480        ipx->ipx_writer = NULL;
6481        ipx->ipx_forced = B_FALSE;
6482        VERIFY(--ipx->ipx_reentry_cnt == 0);
6483        ipx->ipx_ipsq_queued = B_FALSE;
6484        emptied = B_TRUE;
6485 #ifdef  DEBUG
6486        ipx->ipx_depth = 0;
6487 #endif
6488 out:
6489        mutex_exit(&ipx->ipx_lock);
6490        mutex_exit(&ipsq->ipsq_lock);

6492        /*
6493         * If we completely emptied the xop, then wake up any threads waiting
6494         * to enter any of the IPSQ's associated with it.
6495         */
6496        if (emptied) {
6497                xopipsq = ipsq;
6498                do {
6499                        if ((phyi = xopipsq->ipsq_phyint) == NULL)
6500                                continue;

6502                        illv4 = phyi->phyint_illv4;
6503                        illv6 = phyi->phyint_illv6;

6505                        GRAB_ILL_LOCKS(illv4, illv6);
6506                        if (illv4 != NULL)
6507                                cv_broadcast(&illv4->ill_cv);
6508                        if (illv6 != NULL)
6509                                cv_broadcast(&illv6->ill_cv);
6510                        RELEASE_ILL_LOCKS(illv4, illv6);
6511                } while ((xopipsq = xopipsq->ipsq_next) != ipsq);
6512        }
6513        rw_exit(&ipst->ips_ill_g_lock);

6515        /*
6516         * Now that all locks are dropped, exit the IPSQ we left.
6517         */
6518        if (leftipsq != NULL)
6519                ipsq_exit(leftipsq);

6521        return (mp);
6522 }

6524 /*
6525  * Return completion status of previously initiated DLPI operations on
6526  * ills in the purview of an ipsq.
6527  */
6528 static boolean_t
6529 ipsq_dlpi_done(ipsq_t *ipsq)
```



```
6530 {
6531        ipsq_t          *ipsq_start;
6532        phyint_t        *phyi;
6533        ill_t           *ill;

6535        ASSERT(RW_LOCK_HELD(&ipsq->ipsq_ipst->ips_ill_g_lock));
6536        ipsq_start = ipsq;

6538        do {
6539                /*
6540                 * The only current users of this function are ipsq_try_enter
6541                 * and ipsq_enter which have made sure that ipsq_writer is
6542                 * NULL before we reach here. ill_dlpi_pending is modified
6543                 * only by an ipsq writer
6544                 */
6545                ASSERT(ipsq->ipsq_xop->ipx_writer == NULL);
6546                phyi = ipsq->ipsq_phyint;
6547                /*
6548                 * phyi could be NULL if a phyint that is part of an
6549                 * IPMP group is being unplumbed. A more detailed
6550                 * comment is in ipmp_grp_update_kstats()
6551                 */
6552                if (phyi != NULL) {
6553                        ill = phyi->phyint_illv4;
6554                        if (ill != NULL &&
6555                            (ill->ill_dlpi_pending != DL_PRIM_INVAL ||
6556                            ill->ill_arl_dlpi_pending))
6557                                return (B_FALSE);

6559                        ill = phyi->phyint_illv6;
6560                        if (ill != NULL &&
6561                            ill->ill_dlpi_pending != DL_PRIM_INVAL)
6562                                return (B_FALSE);
6563                }

6565        } while ((ipsq = ipsq->ipsq_next) != ipsq_start);

6567        return (B_TRUE);
6568 }

6570 /*
6571  * Enter the ipsq corresponding to ill, by waiting synchronously till
6572  * we can enter the ipsq exclusively. Unless 'force' is used, the ipsq
6573  * will have to drain completely before ipsq_enter returns success.
6574  * ipx_current_ipif will be set if some exclusive op is in progress,
6575  * and the ipsq_exit logic will start the next enqueued op after
6576  * completion of the current op. If 'force' is used, we don't wait
6577  * for the enqueued ops. This is needed when a conn_close wants to
6578  * enter the ipsq and abort an ioctl that is somehow stuck. Unplumb
6579  * of an ill can also use this option. But we dont' use it currently.
6580  */
6581 #define ENTER_SQ_WAIT_TICKS 100
6582 boolean_t
6583 ipsq_enter(ill_t *ill, boolean_t force, int type)
6584 {
6585        ipsq_t  *ipsq;
6586        ipxop_t *ipx;
6587        boolean_t waited_enough = B_FALSE;
6588        ip_stack_t *ipst = ill->ill_ipst;

6590        /*
6591         * Note that the relationship between ill and ipsq is fixed as long as
6592         * the ill is not ILL_CONDEMNED.  Holding ipsq_lock ensures the
6593         * relationship between the IPSQ and xop cannot change.  However,
6594         * since we cannot hold ipsq_lock across the cv_wait(), it may change
6595         * while we're waiting.  We wait on ill_cv and rely on ipsq_exit()
```

```
6596          * waking up all ills in the xop when it becomes available.
6597          */
6598         for (;;) {
6599                 rw_enter(&ipst->ips_ill_g_lock, RW_READER);
6600                 mutex_enter(&ill->ill_lock);
6601                 if (ill->ill_state_flags & ILL_CONDEMNED) {
6602                         mutex_exit(&ill->ill_lock);
6603                         rw_exit(&ipst->ips_ill_g_lock);
6604                         return (B_FALSE);
6605                 }

6607                 ipsq = ill->ill_phyint->phyint_ipsq;
6608                 mutex_enter(&ipsq->ipsq_lock);
6609                 ipx = ipsq->ipsq_xop;
6610                 mutex_enter(&ipx->ipx_lock);

6612                 if (ipx->ipx_writer == NULL && (type == CUR_OP ||
6613                     (ipx->ipx_current_ipif == NULL && ipsq_dlpi_done(ipsq)) ||
6614                     waited_enough))
6615                         break;

6617                 rw_exit(&ipst->ips_ill_g_lock);

6619                 if (!force || ipx->ipx_writer != NULL) {
6620                         mutex_exit(&ipx->ipx_lock);
6621                         mutex_exit(&ipsq->ipsq_lock);
6622                         cv_wait(&ill->ill_cv, &ill->ill_lock);
6623                 } else {
6624                         mutex_exit(&ipx->ipx_lock);
6625                         mutex_exit(&ipsq->ipsq_lock);
6626                         (void) cv_reltimedwait(&ill->ill_cv,
6627                             &ill->ill_lock, ENTER_SQ_WAIT_TICKS, TR_CLOCK_TICK);
6628                         waited_enough = B_TRUE;
6629                 }
6630                 mutex_exit(&ill->ill_lock);
6631         }

6633         ASSERT(ipx->ipx_mphead == NULL && ipx->ipx_mptail == NULL);
6634         ASSERT(ipx->ipx_reentry_cnt == 0);
6635         ipx->ipx_writer = curthread;
6636         ipx->ipx_forced = (ipx->ipx_current_ipif != NULL);
6637         ipx->ipx_reentry_cnt++;
6638 #ifdef DEBUG
6639         ipx->ipx_depth = getpcstack(ipx->ipx_stack, IPX_STACK_DEPTH);
6640 #endif
6641         mutex_exit(&ipx->ipx_lock);
6642         mutex_exit(&ipsq->ipsq_lock);
6643         mutex_exit(&ill->ill_lock);
6644         rw_exit(&ipst->ips_ill_g_lock);

6646         return (B_TRUE);
6647 }

6649 /*
6650  * ipif_set_values() has a constraint that it cannot drop the ips_ill_g_lock
6651  * across the call to the core interface ipsq_try_enter() and hence calls this
6652  * function directly. This is explained more fully in ipif_set_values().
6653  * In order to support the above constraint, ipsq_try_enter is implemented as
6654  * a wrapper that grabs the ips_ill_g_lock and calls this function subsequently
6655  */
6656 static ipsq_t *
6657 ipsq_try_enter_internal(ill_t *ill, queue_t *q, mblk_t *mp, ipsq_func_t func,
6658     int type, boolean_t reentry_ok)
6659 {
6660         ipsq_t  *ipsq;
6661         ipxop_t *ipx;
```



```
6662         ip_stack_t *ipst = ill->ill_ipst;

6664         /*
6665          * lock ordering:
6666          * ill_g_lock -> conn_lock -> ill_lock -> ipsq_lock -> ipx_lock.
6667          *
6668          * ipx of an ipsq can't change when ipsq_lock is held.
6669          */
6670         ASSERT(RW_LOCK_HELD(&ipst->ips_ill_g_lock));
6671         GRAB_CONN_LOCK(q);
6672         mutex_enter(&ill->ill_lock);
6673         ipsq = ill->ill_phyint->phyint_ipsq;
6674         mutex_enter(&ipsq->ipsq_lock);
6675         ipx = ipsq->ipsq_xop;
6676         mutex_enter(&ipx->ipx_lock);

6678         /*
6679          * 1. Enter the ipsq if we are already writer and reentry is ok.
6680          *    (Note: If the caller does not specify reentry_ok then neither
6681          *    'func' nor any of its callees must ever attempt to enter the ipsq
6682          *    again. Otherwise it can lead to an infinite loop
6683          * 2. Enter the ipsq if there is no current writer and this attempted
6684          *    entry is part of the current operation
6685          * 3. Enter the ipsq if there is no current writer and this is a new
6686          *    operation and the operation queue is empty and there is no
6687          *    operation currently in progress and if all previously initiated
6688          *    DLPI operations have completed.
6689          */
6690         if ((ipx->ipx_writer == curthread && reentry_ok) ||
6691             (ipx->ipx_writer == NULL && (type == CUR_OP || (type == NEW_OP &&
6692             !ipx->ipx_ipsq_queued && ipx->ipx_current_ipif == NULL &&
6693             ipsq_dlpi_done(ipsq))))) {
6694                 /* Success. */
6695                 ipx->ipx_reentry_cnt++;
6696                 ipx->ipx_writer = curthread;
6697                 ipx->ipx_forced = B_FALSE;
6698                 mutex_exit(&ipx->ipx_lock);
6699                 mutex_exit(&ipsq->ipsq_lock);
6700                 mutex_exit(&ill->ill_lock);
6701                 RELEASE_CONN_LOCK(q);
6702 #ifdef DEBUG
6703                 ipx->ipx_depth = getpcstack(ipx->ipx_stack, IPX_STACK_DEPTH);
6704 #endif
6705                 return (ipsq);
6706         }

6708         if (func != NULL)
6709                 ipsq_enq(ipsq, q, mp, func, type, ill);

6711         mutex_exit(&ipx->ipx_lock);
6712         mutex_exit(&ipsq->ipsq_lock);
6713         mutex_exit(&ill->ill_lock);
6714         RELEASE_CONN_LOCK(q);
6715         return (NULL);
6716 }

6718 /*
6719  * The ipsq_t (ipsq) is the synchronization data structure used to serialize
6720  * certain critical operations like plumbing (i.e. most set ioctls), etc.
6721  * There is one ipsq per phyint. The ipsq
6722  * serializes exclusive ioctls issued by applications on a per ipsq basis in
6723  * ipsq_xopq_mphead. It also protects against multiple threads executing in
6724  * the ipsq. Responses from the driver pertain to the current ioctl (say a
6725  * DL_BIND_ACK in response to a DL_BIND_REQ initiated as part of bringing
6726  * up the interface) and are enqueued in ipx_mphead.
6727  *
```

```
 * If a thread does not want to reenter the ipsq when it is already writer,
 * it must make sure that the specified reentry point to be called later
 * when the ipsq is empty, nor any code path starting from the specified reentry
 * point must never ever try to enter the ipsq again. Otherwise it can lead
 * to an infinite loop. The reentry point ip_rput_dlpi_writer is an example.
 * When the thread that is currently exclusive finishes, it (ipsq_exit)
 * dequeues the requests waiting to become exclusive in ipx_mphead and calls
 * the reentry point. When the list at ipx_mphead becomes empty ipsq_exit
 * proceeds to dequeue the next ioctl in ipsq_xopq_mphead and start the next
 * ioctl if the current ioctl has completed. If the current ioctl is still
 * in progress it simply returns. The current ioctl could be waiting for
 * a response from another module (the driver or could be waiting for
 * the ipif/ill/ire refcnts to drop to zero. In such a case the ipx_pending_mp
 * and ipx_pending_ipif are set. ipx_current_ipif is set throughout the
 * execution of the ioctl and ipsq_exit does not start the next ioctl unless
 * ipx_current_ipif is NULL which happens only once the ioctl is complete and
 * all associated DLPI operations have completed.
 */

/*
 * Try to enter the IPSQ corresponding to 'ipif' or 'ill' exclusively ('ipif'
 * and 'ill' cannot both be specified).  Returns a pointer to the entered IPSQ
 * on success, or NULL on failure.  The caller ensures ipif/ill is valid by
 * refholding it as necessary.  If the IPSQ cannot be entered and 'func' is
 * non-NULL, then 'func' will be called back with 'q' and 'mp' once the IPSQ
 * can be entered.  If 'func' is NULL, then 'q' and 'mp' are ignored.
 */
ipsq_t *
ipsq_try_enter(ipif_t *ipif, ill_t *ill, queue_t *q, mblk_t *mp,
    ipsq_func_t func, int type, boolean_t reentry_ok)
{
        ip_stack_t      *ipst;
        ipsq_t          *ipsq;

        /* Only 1 of ipif or ill can be specified */
        ASSERT((ipif != NULL) ^ (ill != NULL));

        if (ipif != NULL)
                ill = ipif->ipif_ill;
        ipst = ill->ill_ipst;

        rw_enter(&ipst->ips_ill_g_lock, RW_READER);
        ipsq = ipsq_try_enter_internal(ill, q, mp, func, type, reentry_ok);
        rw_exit(&ipst->ips_ill_g_lock);

        return (ipsq);
}

/*
 * Try to enter the IPSQ corresponding to 'ill' as writer.  The caller ensures
 * ill is valid by refholding it if necessary; we will refrele.  If the IPSQ
 * cannot be entered, the mp is queued for completion.
 */
void
qwriter_ip(ill_t *ill, queue_t *q, mblk_t *mp, ipsq_func_t func, int type,
    boolean_t reentry_ok)
{
        ipsq_t  *ipsq;

        ipsq = ipsq_try_enter(NULL, ill, q, mp, func, type, reentry_ok);

        /*
         * Drop the caller's refhold on the ill.  This is safe since we either
         * entered the IPSQ (and thus are exclusive), or failed to enter the
         * IPSQ, in which case we return without accessing ill anymore.  This
         * is needed because func needs to see the correct refcount.
```



```
         * e.g. removeif can work only then.
         */
        ill_refrele(ill);
        if (ipsq != NULL) {
                (*func)(ipsq, q, mp, NULL);
                ipsq_exit(ipsq);
        }
}

/*
 * Exit the specified IPSQ.  If this is the final exit on it then drain it
 * prior to exiting.  Caller must be writer on the specified IPSQ.
 */
void
ipsq_exit(ipsq_t *ipsq)
{
        mblk_t *mp;
        ipsq_t *mp_ipsq;
        queue_t *q;
        phyint_t *phyi;
        ipsq_func_t func;

        ASSERT(IAM_WRITER_IPSQ(ipsq));

        ASSERT(ipsq->ipsq_xop->ipx_reentry_cnt >= 1);
        if (ipsq->ipsq_xop->ipx_reentry_cnt != 1) {
                ipsq->ipsq_xop->ipx_reentry_cnt--;
                return;
        }

        for (;;) {
                phyi = ipsq->ipsq_phyint;
                mp = ipsq_dq(ipsq);
                mp_ipsq = (mp == NULL) ? NULL : (ipsq_t *)mp->b_next;

                /*
                 * If we've changed to a new IPSQ, and the phyint associated
                 * with the old one has gone away, free the old IPSQ.  Note
                 * that this cannot happen while the IPSQ is in a group.
                 */
                if (mp_ipsq != ipsq && phyi == NULL) {
                        ASSERT(ipsq->ipsq_next == ipsq);
                        ASSERT(ipsq->ipsq_xop == &ipsq->ipsq_ownxop);
                        ipsq_delete(ipsq);
                }

                if (mp == NULL)
                        break;

                q = mp->b_queue;
                func = (ipsq_func_t)mp->b_prev;
                ipsq = mp_ipsq;
                mp->b_next = mp->b_prev = NULL;
                mp->b_queue = NULL;

                /*
                 * If 'q' is an conn queue, it is valid, since we did a
                 * a refhold on the conn at the start of the ioctl.
                 * If 'q' is an ill queue, it is valid, since close of an
                 * ill will clean up its IPSQ.
                 */
                (*func)(ipsq, q, mp, NULL);
        }
}

/*
```

```
 * Used to start any igmp or mld timers that could not be started
 * while holding ill_mcast_lock. The timers can't be started while holding
 * the lock, since mld/igmp_start_timers may need to call untimeout()
 * which can't be done while holding the lock which the timeout handler
 * acquires. Otherwise
 * there could be a deadlock since the timeout handlers
 * mld_timeout_handler_per_ill/igmp_timeout_handler_per_ill also acquire
 * ill_mcast_lock.
 */
void
ill_mcast_timer_start(ip_stack_t *ipst)
{
        int             next;

        mutex_enter(&ipst->ips_igmp_timer_lock);
        next = ipst->ips_igmp_deferred_next;
        ipst->ips_igmp_deferred_next = INFINITY;
        mutex_exit(&ipst->ips_igmp_timer_lock);

        if (next != INFINITY)
                igmp_start_timers(next, ipst);

        mutex_enter(&ipst->ips_mld_timer_lock);
        next = ipst->ips_mld_deferred_next;
        ipst->ips_mld_deferred_next = INFINITY;
        mutex_exit(&ipst->ips_mld_timer_lock);

        if (next != INFINITY)
                mld_start_timers(next, ipst);
}

/*
 * Start the current exclusive operation on 'ipsq'; associate it with 'ipif'
 * and 'ioccmd'.
 */
void
ipsq_current_start(ipsq_t *ipsq, ipif_t *ipif, int ioccmd)
{
        ill_t *ill = ipif->ipif_ill;
        ipxop_t *ipx = ipsq->ipsq_xop;

        ASSERT(IAM_WRITER_IPSQ(ipsq));
        ASSERT(ipx->ipx_current_ipif == NULL);
        ASSERT(ipx->ipx_current_ioctl == 0);

        ipx->ipx_current_done = B_FALSE;
        ipx->ipx_current_ioctl = ioccmd;
        mutex_enter(&ipx->ipx_lock);
        ipx->ipx_current_ipif = ipif;
        mutex_exit(&ipx->ipx_lock);

        /*
         * Set IPIF_CHANGING on one or more ipifs associated with the
         * current exclusive operation.  IPIF_CHANGING prevents any new
         * references to the ipif (so that the references will eventually
         * drop to zero) and also prevents any "get" operations (e.g.,
         * SIOCGLIFFLAGS) from being able to access the ipif until the
         * operation has completed and the ipif is again in a stable state.
         *
         * For ioctls, IPIF_CHANGING is set on the ipif associated with the
         * ioctl.  For internal operations (where ioccmd is zero), all ipifs
         * on the ill are marked with IPIF_CHANGING since it's unclear which
         * ipifs will be affected.
         *
         * Note that SIOCLIFREMOVEIF is a special case as it sets
         * IPIF_CONDEMNED internally after identifying the right ipif to
```



```
         * operate on.
         */
        switch (ioccmd) {
        case SIOCLIFREMOVEIF:
                break;
        case 0:
                mutex_enter(&ill->ill_lock);
                ipif = ipif->ipif_ill->ill_ipif;
                for (; ipif != NULL; ipif = ipif->ipif_next)
                        ipif->ipif_state_flags |= IPIF_CHANGING;
                mutex_exit(&ill->ill_lock);
                break;
        default:
                mutex_enter(&ill->ill_lock);
                ipif->ipif_state_flags |= IPIF_CHANGING;
                mutex_exit(&ill->ill_lock);
        }
}

/*
 * Finish the current exclusive operation on 'ipsq'.  Usually, this will allow
 * the next exclusive operation to begin once we ipsq_exit().  However, if
 * pending DLPI operations remain, then we will wait for the queue to drain
 * before allowing the next exclusive operation to begin.  This ensures that
 * DLPI operations from one exclusive operation are never improperly processed
 * as part of a subsequent exclusive operation.
 */
void
ipsq_current_finish(ipsq_t *ipsq)
{
        ipxop_t *ipx = ipsq->ipsq_xop;
        t_uscalar_t dlpi_pending = DL_PRIM_INVAL;
        ipif_t  *ipif = ipx->ipx_current_ipif;

        ASSERT(IAM_WRITER_IPSQ(ipsq));

        /*
         * For SIOCLIFREMOVEIF, the ipif has been already been blown away
         * (but in that case, IPIF_CHANGING will already be clear and no
         * pending DLPI messages can remain).
         */
        if (ipx->ipx_current_ioctl != SIOCLIFREMOVEIF) {
                ill_t *ill = ipif->ipif_ill;

                mutex_enter(&ill->ill_lock);
                dlpi_pending = ill->ill_dlpi_pending;
                if (ipx->ipx_current_ioctl == 0) {
                        ipif = ill->ill_ipif;
                        for (; ipif != NULL; ipif = ipif->ipif_next)
                                ipif->ipif_state_flags &= ~IPIF_CHANGING;
                } else {
                        ipif->ipif_state_flags &= ~IPIF_CHANGING;
                }
                mutex_exit(&ill->ill_lock);
        }

        ASSERT(!ipx->ipx_current_done);
        ipx->ipx_current_done = B_TRUE;
        ipx->ipx_current_ioctl = 0;
        if (dlpi_pending == DL_PRIM_INVAL) {
                mutex_enter(&ipx->ipx_lock);
                ipx->ipx_current_ipif = NULL;
                mutex_exit(&ipx->ipx_lock);
        }
}
```

```
6992 /*
6993  * The ill is closing. Flush all messages on the ipsq that originated
6994  * from this ill. Usually there wont' be any messages on the ipsq_xopq_mphead
6995  * for this ill since ipsq_enter could not have entered until then.
6996  * New messages can't be queued since the CONDEMNED flag is set.
6997  */
6998 static void
6999 ipsq_flush(ill_t *ill)
7000 {
7001         queue_t *q;
7002         mblk_t  *prev;
7003         mblk_t  *mp;
7004         mblk_t  *mp_next;
7005         ipxop_t *ipx = ill->ill_phyint->phyint_ipsq->ipsq_xop;

7007         ASSERT(IAM_WRITER_ILL(ill));

7009         /*
7010          * Flush any messages sent up by the driver.
7011          */
7012         mutex_enter(&ipx->ipx_lock);
7013         for (prev = NULL, mp = ipx->ipx_mphead; mp != NULL; mp = mp_next) {
7014                 mp_next = mp->b_next;
7015                 q = mp->b_queue;
7016                 if (q == ill->ill_rq || q == ill->ill_wq) {
7017                         /* dequeue mp */
7018                         if (prev == NULL)
7019                                 ipx->ipx_mphead = mp->b_next;
7020                         else
7021                                 prev->b_next = mp->b_next;
7022                         if (ipx->ipx_mptail == mp) {
7023                                 ASSERT(mp_next == NULL);
7024                                 ipx->ipx_mptail = prev;
7025                         }
7026                         inet_freemsg(mp);
7027                 } else {
7028                         prev = mp;
7029                 }
7030         }
7031         mutex_exit(&ipx->ipx_lock);
7032         (void) ipsq_pending_mp_cleanup(ill, NULL);
7033         ipsq_xopq_mp_cleanup(ill, NULL);
7034 }

7036 /*
7037  * Parse an ifreq or lifreq struct coming down ioctls and refhold
7038  * and return the associated ipif.
7039  * Return value:
7040  *      Non zero: An error has occurred. ci may not be filled out.
7041  *      zero : ci is filled out with the ioctl cmd in ci.ci_name, and
7042  *      a held ipif in ci.ci_ipif.
7043  */
7044 int
7045 ip_extract_lifreq(queue_t *q, mblk_t *mp, const ip_ioctl_cmd_t *ipip,
7046     cmd_info_t *ci)
7047 {
7048         char            *name;
7049         struct ifreq    *ifr;
7050         struct lifreq    *lifr;
7051         ipif_t          *ipif = NULL;
7052         ill_t           *ill;
7053         conn_t          *connp;
7054         boolean_t       isv6;
7055         int             err;
7056         mblk_t          *mp1;
7057         zoneid_t        zoneid;
```



```
7058         ip_stack_t      *ipst;

7060         if (q->q_next != NULL) {
7061                 ill = (ill_t *)q->q_ptr;
7062                 isv6 = ill->ill_isv6;
7063                 connp = NULL;
7064                 zoneid = ALL_ZONES;
7065                 ipst = ill->ill_ipst;
7066         } else {
7067                 ill = NULL;
7068                 connp = Q_TO_CONN(q);
7069                 isv6 = (connp->conn_family == AF_INET6);
7070                 zoneid = connp->conn_zoneid;
7071                 if (zoneid == GLOBAL_ZONEID) {
7072                         /* global zone can access ipifs in all zones */
7073                         zoneid = ALL_ZONES;
7074                 }
7075                 ipst = connp->conn_netstack->netstack_ip;
7076         }

7078         /* Has been checked in ip_wput_nondata */
7079         mp1 = mp->b_cont->b_cont;

7081         if (ipip->ipi_cmd_type == IF_CMD) {
7082                 /* This a old style SIOC[GS]IF* command */
7083                 ifr = (struct ifreq *)mp1->b_rptr;
7084                 /*
7085                  * Null terminate the string to protect against buffer
7086                  * overrun. String was generated by user code and may not
7087                  * be trusted.
7088                  */
7089                 ifr->ifr_name[IFNAMSIZ - 1] = '\0';
7090                 name = ifr->ifr_name;
7091                 ci->ci_sin = (sin_t *)&ifr->ifr_addr;
7092                 ci->ci_sin6 = NULL;
7093                 ci->ci_lifr = (struct lifreq *)ifr;
7094         } else {
7095                 /* This a new style SIOC[GS]LIF* command */
7096                 ASSERT(ipip->ipi_cmd_type == LIF_CMD);
7097                 lifr = (struct lifreq *)mp1->b_rptr;
7098                 /*
7099                  * Null terminate the string to protect against buffer
7100                  * overrun. String was generated by user code and may not
7101                  * be trusted.
7102                  */
7103                 lifr->lifr_name[LIFNAMSIZ - 1] = '\0';
7104                 name = lifr->lifr_name;
7105                 ci->ci_sin = (sin_t *)&lifr->lifr_addr;
7106                 ci->ci_sin6 = (sin6_t *)&lifr->lifr_addr;
7107                 ci->ci_lifr = lifr;
7108         }

7110         if (ipip->ipi_cmd == SIOCSLIFNAME) {
7111                 /*
7112                  * The ioctl will be failed if the ioctl comes down
7113                  * an conn stream
7114                  */
7115                 if (ill == NULL) {
7116                         /*
7117                          * Not an ill queue, return EINVAL same as the
7118                          * old error code.
7119                          */
7120                         return (ENXIO);
7121                 }
7122                 ipif = ill->ill_ipif;
7123                 ipif_refhold(ipif);
```

```
7124         } else {
7125                 /*
7126                  * Ensure that ioctls don't see any internal state changes
7127                  * caused by set ioctls by deferring them if IPIF_CHANGING is
7128                  * set.
7129                  */
7130                 ipif = ipif_lookup_on_name_async(name, mi_strlen(name),
7131                     isv6, zoneid, q, mp, ip_process_ioctl, &err, ipst);
7132                 if (ipif == NULL) {
7133                         if (err == EINPROGRESS)
7134                                 return (err);
7135                         err = 0;        /* Ensure we don't use it below */
7136                 }
7137         }
7139         /*
7140          * Old style [GS]IFCMD does not admit IPv6 ipif
7141          */
7142         if (ipif != NULL && ipif->ipif_isv6 && ipip->ipi_cmd_type == IF_CMD) {
7143                 ipif_refrele(ipif);
7144                 return (ENXIO);
7145         }
7147         if (ipif == NULL && ill != NULL && ill->ill_ipif != NULL &&
7148             name[0] == '\0') {
7149                 /*
7150                  * Handle a or a SIOC?IF* with a null name
7151                  * during plumb (on the ill queue before the I_PLINK).
7152                  */
7153                 ipif = ill->ill_ipif;
7154                 ipif_refhold(ipif);
7155         }
7157         if (ipif == NULL)
7158                 return (ENXIO);
7160         DTRACE_PROBE4(ipif__ioctl, char *, "ip_extract_lifreq",
7161             int, ipip->ipi_cmd, ill_t *, ipif->ipif_ill, ipif_t *, ipif);
7163         ci->ci_ipif = ipif;
7164         return (0);
7165 }
7167 /*
7168  * Return the total number of ipifs.
7169  */
7170 static uint_t
7171 ip_get_numifs(zoneid_t zoneid, ip_stack_t *ipst)
7172 {
7173         uint_t numifs = 0;
7174         ill_t   *ill;
7175         ill_walk_context_t      ctx;
7176         ipif_t  *ipif;
7178         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
7179         ill = ILL_START_WALK_V4(&ctx, ipst);
7180         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
7181                 if (IS_UNDER_IPMP(ill))
7182                         continue;
7183                 for (ipif = ill->ill_ipif; ipif != NULL;
7184                     ipif = ipif->ipif_next) {
7185                         if (ipif->ipif_zoneid == zoneid ||
7186                             ipif->ipif_zoneid == ALL_ZONES)
7187                                 numifs++;
7188                 }
7189         }
```



```
7190         rw_exit(&ipst->ips_ill_g_lock);
7191         return (numifs);
7192 }
7194 /*
7195  * Return the total number of ipifs.
7196  */
7197 static uint_t
7198 ip_get_numlifs(int family, int lifn_flags, zoneid_t zoneid, ip_stack_t *ipst)
7199 {
7200         uint_t numifs = 0;
7201         ill_t   *ill;
7202         ipif_t  *ipif;
7203         ill_walk_context_t      ctx;
7205         ip1dbg(("ip_get_numlifs(%d %u %d)\n", family, lifn_flags, (int)zoneid));
7207         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
7208         if (family == AF_INET)
7209                 ill = ILL_START_WALK_V4(&ctx, ipst);
7210         else if (family == AF_INET6)
7211                 ill = ILL_START_WALK_V6(&ctx, ipst);
7212         else
7213                 ill = ILL_START_WALK_ALL(&ctx, ipst);
7215         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
7216                 if (IS_UNDER_IPMP(ill) && !(lifn_flags & LIFC_UNDER_IPMP))
7217                         continue;
7219                 for (ipif = ill->ill_ipif; ipif != NULL;
7220                     ipif = ipif->ipif_next) {
7221                         if ((ipif->ipif_flags & IPIF_NOXMIT) &&
7222                             !(lifn_flags & LIFC_NOXMIT))
7223                                 continue;
7224                         if ((ipif->ipif_flags & IPIF_TEMPORARY) &&
7225                             !(lifn_flags & LIFC_TEMPORARY))
7226                                 continue;
7227                         if (((ipif->ipif_flags &
7228                             (IPIF_NOXMIT|IPIF_NOLOCAL|
7229                             IPIF_DEPRECATED)) ||
7230                             IS_LOOPBACK(ill) ||
7231                             !(ipif->ipif_flags & IPIF_UP)) &&
7232                             (lifn_flags & LIFC_EXTERNAL_SOURCE))
7233                                 continue;
7235                         if (zoneid != ipif->ipif_zoneid &&
7236                             ipif->ipif_zoneid != ALL_ZONES &&
7237                             (zoneid != GLOBAL_ZONEID ||
7238                             !(lifn_flags & LIFC_ALLZONES)))
7239                                 continue;
7241                         numifs++;
7242                 }
7243         }
7244         rw_exit(&ipst->ips_ill_g_lock);
7245         return (numifs);
7246 }
7248 uint_t
7249 ip_get_lifsrcofnum(ill_t *ill)
7250 {
7251         uint_t numifs = 0;
7252         ill_t   *ill_head = ill;
7253         ip_stack_t      *ipst = ill->ill_ipst;
7255         /*
```

```
7256          * ill_g_usesrc_lock protects ill_usesrc_grp_next, for example, some
7257          * other thread may be trying to relink the ILLs in this usesrc group
7258          * and adjusting the ill_usesrc_grp_next pointers
7259          */
7260         rw_enter(&ipst->ips_ill_g_usesrc_lock, RW_READER);
7261         if ((ill->ill_usesrc_ifindex == 0) &&
7262             (ill->ill_usesrc_grp_next != NULL)) {
7263                 for (; (ill != NULL) && (ill->ill_usesrc_grp_next != ill_head);
7264                     ill = ill->ill_usesrc_grp_next)
7265                         numifs++;
7266         }
7267         rw_exit(&ipst->ips_ill_g_usesrc_lock);

7269         return (numifs);
7270 }

7272 /* Null values are passed in for ipif, sin, and ifreq */
7273 /* ARGSUSED */
7274 int
7275 ip_sioctl_get_ifnum(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7276     mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7277 {
7278         int *nump;
7279         conn_t *connp = Q_TO_CONN(q);

7281         ASSERT(q->q_next == NULL); /* not a valid ioctl for ip as a module */

7283         /* Existence of b_cont->b_cont checked in ip_wput_nondata */
7284         nump = (int *)mp->b_cont->b_cont->b_rptr;

7286         *nump = ip_get_numifs(connp->conn_zoneid,
7287             connp->conn_netstack->netstack_ip);
7288         ip1dbg(("ip_sioctl_get_ifnum numifs %d", *nump));
7289         return (0);
7290 }

7292 /* Null values are passed in for ipif, sin, and ifreq */
7293 /* ARGSUSED */
7294 int
7295 ip_sioctl_get_lifnum(ipif_t *dummy_ipif, sin_t *dummy_sin,
7296     queue_t *q, mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7297 {
7298         struct lifnum *lifn;
7299         mblk_t  *mp1;
7300         conn_t *connp = Q_TO_CONN(q);

7302         ASSERT(q->q_next == NULL); /* not a valid ioctl for ip as a module */

7304         /* Existence checked in ip_wput_nondata */
7305         mp1 = mp->b_cont->b_cont;

7307         lifn = (struct lifnum *)mp1->b_rptr;
7308         switch (lifn->lifn_family) {
7309         case AF_UNSPEC:
7310         case AF_INET:
7311         case AF_INET6:
7312                 break;
7313         default:
7314                 return (EAFNOSUPPORT);
7315         }

7317         lifn->lifn_count = ip_get_numlifs(lifn->lifn_family, lifn->lifn_flags,
7318             connp->conn_zoneid, connp->conn_netstack->netstack_ip);
7319         ip1dbg(("ip_sioctl_get_lifnum numifs %d", lifn->lifn_count));
7320         return (0);
7321 }
```



```
7323 /* ARGSUSED */
7324 int
7325 ip_sioctl_get_ifconf(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7326     mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7327 {
7328         STRUCT_HANDLE(ifconf, ifc);
7329         mblk_t *mp1;
7330         struct iocblk *iocp;
7331         struct ifreq *ifr;
7332         ill_walk_context_t      ctx;
7333         ill_t   *ill;
7334         ipif_t  *ipif;
7335         struct sockaddr_in *sin;
7336         int32_t ifclen;
7337         zoneid_t zoneid;
7338         ip_stack_t *ipst = CONNQ_TO_IPST(q);

7340         ASSERT(q->q_next == NULL); /* not valid ioctls for ip as a module */

7342         ip1dbg(("ip_sioctl_get_ifconf"));
7343         /* Existence verified in ip_wput_nondata */
7344         mp1 = mp->b_cont->b_cont;
7345         iocp = (struct iocblk *)mp->b_rptr;
7346         zoneid = Q_TO_CONN(q)->conn_zoneid;

7348         /*
7349          * The original SIOCGIFCONF passed in a struct ifconf which specified
7350          * the user buffer address and length into which the list of struct
7351          * ifreqs was to be copied.  Since AT&T Streams does not seem to
7352          * allow M_COPYOUT to be used in conjunction with I_STR IOCTLS,
7353          * the SIOCGIFCONF operation was redefined to simply provide
7354          * a large output buffer into which we are supposed to jam the ifreq
7355          * array.  The same ioctl command code was used, despite the fact that
7356          * both the applications and the kernel code had to change, thus making
7357          * it impossible to support both interfaces.
7358          *
7359          * For reasons not good enough to try to explain, the following
7360          * algorithm is used for deciding what to do with one of these:
7361          * If the IOCTL comes in as an I_STR, it is assumed to be of the new
7362          * form with the output buffer coming down as the continuation message.
7363          * If it arrives as a TRANSPARENT IOCTL, it is assumed to be old style,
7364          * and we have to copy in the ifconf structure to find out how big the
7365          * output buffer is and where to copy out to.  Sure no problem...
7366          *
7367          */
7368         STRUCT_SET_HANDLE(ifc, iocp->ioc_flag, NULL);
7369         if ((mp1->b_wptr - mp1->b_rptr) == STRUCT_SIZE(ifc)) {
7370                 int numifs = 0;
7371                 size_t ifc_bufsize;

7373                 /*
7374                  * Must be (better be!) continuation of a TRANSPARENT
7375                  * IOCTL.  We just copied in the ifconf structure.
7376                  */
7377                 STRUCT_SET_HANDLE(ifc, iocp->ioc_flag,
7378                     (struct ifconf *)mp1->b_rptr);

7380                 /*
7381                  * Allocate a buffer to hold requested information.
7382                  *
7383                  * If ifc_len is larger than what is needed, we only
7384                  * allocate what we will use.
7385                  *
7386                  * If ifc_len is smaller than what is needed, return
7387                  * EINVAL.
```

```
7388                 *
7389                 * XXX: the ill_t structure can hava 2 counters, for
7390                 * v4 and v6 (not just ill_ipif_up_count) to store the
7391                 * number of interfaces for a device, so we don't need
7392                 * to count them here...
7393                 */
7394                numifs = ip_get_numifs(zoneid, ipst);

7396                ifclen = STRUCT_FGET(ifc, ifc_len);
7397                ifc_bufsize = numifs * sizeof (struct ifreq);
7398                if (ifc_bufsize > ifclen) {
7399                        if (iocp->ioc_cmd == O_SIOCGIFCONF) {
7400                                /* old behaviour */
7401                                return (EINVAL);
7402                        } else {
7403                                ifc_bufsize = ifclen;
7404                        }
7405                }

7407                mp1 = mi_copyout_alloc(q, mp,
7408                    STRUCT_FGETP(ifc, ifc_buf), ifc_bufsize, B_FALSE);
7409                if (mp1 == NULL)
7410                        return (ENOMEM);

7412                mp1->b_wptr = mp1->b_rptr + ifc_bufsize;
7413        }
7414        bzero(mp1->b_rptr, mp1->b_wptr - mp1->b_rptr);
7415        /*
7416         * the SIOCGIFCONF ioctl only knows about
7417         * IPv4 addresses, so don't try to tell
7418         * it about interfaces with IPv6-only
7419         * addresses. (Last parm 'isv6' is B_FALSE)
7420         */

7422        ifr = (struct ifreq *)mp1->b_rptr;

7424        rw_enter(&ipst->ips_ill_g_lock, RW_READER);
7425        ill = ILL_START_WALK_V4(&ctx, ipst);
7426        for (; ill != NULL; ill = ill_next(&ctx, ill)) {
7427                if (IS_UNDER_IPMP(ill))
7428                        continue;
7429                for (ipif = ill->ill_ipif; ipif != NULL;
7430                    ipif = ipif->ipif_next) {
7431                        if (zoneid != ipif->ipif_zoneid &&
7432                            ipif->ipif_zoneid != ALL_ZONES)
7433                                continue;
7434                        if ((uchar_t *)&ifr[1] > mp1->b_wptr) {
7435                                if (iocp->ioc_cmd == O_SIOCGIFCONF) {
7436                                        /* old behaviour */
7437                                        rw_exit(&ipst->ips_ill_g_lock);
7438                                        return (EINVAL);
7439                                } else {
7440                                        goto if_copydone;
7441                                }
7442                        }
7443                        ipif_get_name(ipif, ifr->ifr_name,
7444                            sizeof (ifr->ifr_name));
7445                        sin = (sin_t *)&ifr->ifr_addr;
7446                        *sin = sin_null;
7447                        sin->sin_family = AF_INET;
7448                        sin->sin_addr.s_addr = ipif->ipif_lcl_addr;
7449                        ifr++;
7450                }
7451        }
7452 if_copydone:
7453        rw_exit(&ipst->ips_ill_g_lock);
```



```
7454        mp1->b_wptr = (uchar_t *)ifr;

7456        if (STRUCT_BUF(ifc) != NULL) {
7457                STRUCT_FSET(ifc, ifc_len,
7458                    (int)((uchar_t *)ifr - mp1->b_rptr));
7459        }
7460        return (0);
7461 }

7463 /*
7464  * Get the interfaces using the address hosted on the interface passed in,
7465  * as a source adddress
7466  */
7467 /* ARGSUSED */
7468 int
7469 ip_sioctl_get_lifsrcof(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7470     mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7471 {
7472        mblk_t *mp1;
7473        ill_t   *ill, *ill_head;
7474        ipif_t  *ipif, *orig_ipif;
7475        int     numlifs = 0;
7476        size_t  lifs_bufsize, lifsmaxlen;
7477        struct  lifreq *lifr;
7478        struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
7479        uint_t  ifindex;
7480        zoneid_t zoneid;
7481        boolean_t isv6 = B_FALSE;
7482        struct  sockaddr_in     *sin;
7483        struct  sockaddr_in6    *sin6;
7484        STRUCT_HANDLE(lifsrcof, lifs);
7485        ip_stack_t              *ipst;

7487        ipst = CONNQ_TO_IPST(q);

7489        ASSERT(q->q_next == NULL);

7491        zoneid = Q_TO_CONN(q)->conn_zoneid;

7493        /* Existence verified in ip_wput_nondata */
7494        mp1 = mp->b_cont->b_cont;

7496        /*
7497         * Must be (better be!) continuation of a TRANSPARENT
7498         * IOCTL.  We just copied in the lifsrcof structure.
7499         */
7500        STRUCT_SET_HANDLE(lifs, iocp->ioc_flag,
7501            (struct lifsrcof *)mp1->b_rptr);

7503        if (MBLKL(mp1) != STRUCT_SIZE(lifs))
7504                return (EINVAL);

7506        ifindex = STRUCT_FGET(lifs, lifs_ifindex);
7507        isv6 = (Q_TO_CONN(q))->conn_family == AF_INET6;
7508        ipif = ipif_lookup_on_ifindex(ifindex, isv6, zoneid, ipst);
7509        if (ipif == NULL) {
7510                ip1dbg(("ip_sioctl_get_lifsrcof: no ipif for ifindex %d\n",
7511                    ifindex));
7512                return (ENXIO);
7513        }

7515        /* Allocate a buffer to hold requested information */
7516        numlifs = ip_get_lifsrcofnum(ipif->ipif_ill);
7517        lifs_bufsize = numlifs * sizeof (struct lifreq);
7518        lifsmaxlen =  STRUCT_FGET(lifs, lifs_maxlen);
7519        /* The actual size needed is always returned in lifs_len */
```

```
7520         STRUCT_FSET(lifs, lifs_len, lifs_bufsize);

7522         /* If the amount we need is more than what is passed in, abort */
7523         if (lifs_bufsize > lifsmaxlen || lifs_bufsize == 0) {
7524                 ipif_refrele(ipif);
7525                 return (0);
7526         }

7528         mp1 = mi_copyout_alloc(q, mp,
7529             STRUCT_FGETP(lifs, lifs_buf), lifs_bufsize, B_FALSE);
7530         if (mp1 == NULL) {
7531                 ipif_refrele(ipif);
7532                 return (ENOMEM);
7533         }

7535         mp1->b_wptr = mp1->b_rptr + lifs_bufsize;
7536         bzero(mp1->b_rptr, lifs_bufsize);

7538         lifr = (struct lifreq *)mp1->b_rptr;

7540         ill = ill_head = ipif->ipif_ill;
7541         orig_ipif = ipif;

7543         /* ill_g_usesrc_lock protects ill_usesrc_grp_next */
7544         rw_enter(&ipst->ips_ill_g_usesrc_lock, RW_READER);
7545         rw_enter(&ipst->ips_ill_g_lock, RW_READER);

7547         ill = ill->ill_usesrc_grp_next; /* start from next ill */
7548         for (; (ill != NULL) && (ill != ill_head);
7549             ill = ill->ill_usesrc_grp_next) {

7551                 if ((uchar_t *)&lifr[1] > mp1->b_wptr)
7552                         break;

7554                 ipif = ill->ill_ipif;
7555                 ipif_get_name(ipif, lifr->lifr_name, sizeof (lifr->lifr_name));
7556                 if (ipif->ipif_isv6) {
7557                         sin6 = (sin6_t *)&lifr->lifr_addr;
7558                         *sin6 = sin6_null;
7559                         sin6->sin6_family = AF_INET6;
7560                         sin6->sin6_addr = ipif->ipif_v6lcl_addr;
7561                         lifr->lifr_addrlen = ip_mask_to_plen_v6(
7562                             &ipif->ipif_v6net_mask);
7563                 } else {
7564                         sin = (sin_t *)&lifr->lifr_addr;
7565                         *sin = sin_null;
7566                         sin->sin_family = AF_INET;
7567                         sin->sin_addr.s_addr = ipif->ipif_lcl_addr;
7568                         lifr->lifr_addrlen = ip_mask_to_plen(
7569                             ipif->ipif_net_mask);
7570                 }
7571                 lifr++;
7572         }
7573         rw_exit(&ipst->ips_ill_g_lock);
7574         rw_exit(&ipst->ips_ill_g_usesrc_lock);
7575         ipif_refrele(orig_ipif);
7576         mp1->b_wptr = (uchar_t *)lifr;
7577         STRUCT_FSET(lifs, lifs_len, (int)((uchar_t *)lifr - mp1->b_rptr));

7579         return (0);
7580 }

7582 /* ARGSUSED */
7583 int
7584 ip_sioctl_get_lifconf(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7585     mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
```



```
7586 {
7587         mblk_t *mp1;
7588         int     list;
7589         ill_t   *ill;
7590         ipif_t  *ipif;
7591         int     flags;
7592         int     numlifs = 0;
7593         size_t  lifc_bufsize;
7594         struct  lifreq *lifr;
7595         sa_family_t     family;
7596         struct  sockaddr_in     *sin;
7597         struct  sockaddr_in6    *sin6;
7598         ill_walk_context_t      ctx;
7599         struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
7600         int32_t lifclen;
7601         zoneid_t zoneid;
7602         STRUCT_HANDLE(lifconf, lifc);
7603         ip_stack_t *ipst = CONNQ_TO_IPST(q);

7605         ip1dbg(("ip_sioctl_get_lifconf"));

7607         ASSERT(q->q_next == NULL);

7609         zoneid = Q_TO_CONN(q)->conn_zoneid;

7611         /* Existence verified in ip_wput_nondata */
7612         mp1 = mp->b_cont->b_cont;

7614         /*
7615          * An extended version of SIOCGIFCONF that takes an
7616          * additional address family and flags field.
7617          * AF_UNSPEC retrieve both IPv4 and IPv6.
7618          * Unless LIFC_NOXMIT is specified the IPIF_NOXMIT
7619          * interfaces are omitted.
7620          * Similarly, IPIF_TEMPORARY interfaces are omitted
7621          * unless LIFC_TEMPORARY is specified.
7622          * If LIFC_EXTERNAL_SOURCE is specified, IPIF_NOXMIT,
7623          * IPIF_NOLOCAL, PHYI_LOOPBACK, IPIF_DEPRECATED and
7624          * not IPIF_UP interfaces are omitted. LIFC_EXTERNAL_SOURCE
7625          * has priority over LIFC_NOXMIT.
7626          */
7627         STRUCT_SET_HANDLE(lifc, iocp->ioc_flag, NULL);

7629         if ((mp1->b_wptr - mp1->b_rptr) != STRUCT_SIZE(lifc))
7630                 return (EINVAL);

7632         /*
7633          * Must be (better be!) continuation of a TRANSPARENT
7634          * IOCTL.  We just copied in the lifconf structure.
7635          */
7636         STRUCT_SET_HANDLE(lifc, iocp->ioc_flag, (struct lifconf *)mp1->b_rptr);

7638         family = STRUCT_FGET(lifc, lifc_family);
7639         flags = STRUCT_FGET(lifc, lifc_flags);

7641         switch (family) {
7642         case AF_UNSPEC:
7643                 /*
7644                  * walk all ILL's.
7645                  */
7646                 list = MAX_G_HEADS;
7647                 break;
7648         case AF_INET:
7649                 /*
7650                  * walk only IPV4 ILL's.
7651                  */
```

```
7652                 list = IP_V4_G_HEAD;
7653                 break;
7654         case AF_INET6:
7655                 /*
7656                  * walk only IPV6 ILL's.
7657                  */
7658                 list = IP_V6_G_HEAD;
7659                 break;
7660         default:
7661                 return (EAFNOSUPPORT);
7662         }

7664         /*
7665          * Allocate a buffer to hold requested information.
7666          *
7667          * If lifc_len is larger than what is needed, we only
7668          * allocate what we will use.
7669          *
7670          * If lifc_len is smaller than what is needed, return
7671          * EINVAL.
7672          */
7673         numlifs = ip_get_numlifs(family, flags, zoneid, ipst);
7674         lifc_bufsize = numlifs * sizeof (struct lifreq);
7675         lifclen = STRUCT_FGET(lifc, lifc_len);
7676         if (lifc_bufsize > lifclen) {
7677                 if (iocp->ioc_cmd == O_SIOCGLIFCONF)
7678                         return (EINVAL);
7679                 else
7680                         lifc_bufsize = lifclen;
7681         }

7683         mp1 = mi_copyout_alloc(q, mp,
7684             STRUCT_FGETP(lifc, lifc_buf), lifc_bufsize, B_FALSE);
7685         if (mp1 == NULL)
7686                 return (ENOMEM);

7688         mp1->b_wptr = mp1->b_rptr + lifc_bufsize;
7689         bzero(mp1->b_rptr, mp1->b_wptr - mp1->b_rptr);

7691         lifr = (struct lifreq *)mp1->b_rptr;

7693         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
7694         ill = ill_first(list, list, &ctx, ipst);
7695         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
7696                 if (IS_UNDER_IPMP(ill) && !(flags & LIFC_UNDER_IPMP))
7697                         continue;

7699                 for (ipif = ill->ill_ipif; ipif != NULL;
7700                     ipif = ipif->ipif_next) {
7701                         if ((ipif->ipif_flags & IPIF_NOXMIT) &&
7702                             !(flags & LIFC_NOXMIT))
7703                                 continue;

7705                         if ((ipif->ipif_flags & IPIF_TEMPORARY) &&
7706                             !(flags & LIFC_TEMPORARY))
7707                                 continue;

7709                         if (((ipif->ipif_flags &
7710                             (IPIF_NOXMIT|IPIF_NOLOCAL|
7711                             IPIF_DEPRECATED)) ||
7712                             IS_LOOPBACK(ill) ||
7713                             !(ipif->ipif_flags & IPIF_UP)) &&
7714                             (flags & LIFC_EXTERNAL_SOURCE))
7715                                 continue;

7717                         if (zoneid != ipif->ipif_zoneid &&
```



```
7718                             ipif->ipif_zoneid != ALL_ZONES &&
7719                             (zoneid != GLOBAL_ZONEID ||
7720                             !(flags & LIFC_ALLZONES)))
7721                                 continue;

7723                         if ((uchar_t *)&lifr[1] > mp1->b_wptr) {
7724                                 if (iocp->ioc_cmd == O_SIOCGLIFCONF) {
7725                                         rw_exit(&ipst->ips_ill_g_lock);
7726                                         return (EINVAL);
7727                                 } else {
7728                                         goto lif_copydone;
7729                                 }
7730                         }

7732                         ipif_get_name(ipif, lifr->lifr_name,
7733                             sizeof (lifr->lifr_name));
7734                         lifr->lifr_type = ill->ill_type;
7735                         if (ipif->ipif_isv6) {
7736                                 sin6 = (sin6_t *)&lifr->lifr_addr;
7737                                 *sin6 = sin6_null;
7738                                 sin6->sin6_family = AF_INET6;
7739                                 sin6->sin6_addr =
7740                                     ipif->ipif_v6lcl_addr;
7741                                 lifr->lifr_addrlen =
7742                                     ip_mask_to_plen_v6(
7743                                     &ipif->ipif_v6net_mask);
7744                         } else {
7745                                 sin = (sin_t *)&lifr->lifr_addr;
7746                                 *sin = sin_null;
7747                                 sin->sin_family = AF_INET;
7748                                 sin->sin_addr.s_addr =
7749                                     ipif->ipif_lcl_addr;
7750                                 lifr->lifr_addrlen =
7751                                     ip_mask_to_plen(
7752                                     ipif->ipif_net_mask);
7753                         }
7754                         lifr++;
7755                 }
7756         }
7757 lif_copydone:
7758         rw_exit(&ipst->ips_ill_g_lock);

7760         mp1->b_wptr = (uchar_t *)lifr;
7761         if (STRUCT_BUF(lifc) != NULL) {
7762                 STRUCT_FSET(lifc, lifc_len,
7763                     (int)((uchar_t *)lifr - mp1->b_rptr));
7764         }
7765         return (0);
7766 }

7768 static void
7769 ip_sioctl_ip6addrpolicy(queue_t *q, mblk_t *mp)
7770 {
7771         ip6_asp_t *table;
7772         size_t table_size;
7773         mblk_t *data_mp;
7774         struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
7775         ip_stack_t      *ipst;

7777         if (q->q_next == NULL)
7778                 ipst = CONNQ_TO_IPST(q);
7779         else
7780                 ipst = ILLQ_TO_IPST(q);

7782         /* These two ioctls are I_STR only */
7783         if (iocp->ioc_count == TRANSPARENT) {
```

```
7784                miocnak(q, mp, 0, EINVAL);
7785                return;
7786        }

7788        data_mp = mp->b_cont;
7789        if (data_mp == NULL) {
7790                /* The user passed us a NULL argument */
7791                table = NULL;
7792                table_size = iocp->ioc_count;
7793        } else {
7794                /*
7795                 * The user provided a table.  The stream head
7796                 * may have copied in the user data in chunks,
7797                 * so make sure everything is pulled up
7798                 * properly.
7799                 */
7800                if (MBLKL(data_mp) < iocp->ioc_count) {
7801                        mblk_t *new_data_mp;
7802                        if ((new_data_mp = msgpullup(data_mp, -1)) ==
7803                            NULL) {
7804                                miocnak(q, mp, 0, ENOMEM);
7805                                return;
7806                        }
7807                        freemsg(data_mp);
7808                        data_mp = new_data_mp;
7809                        mp->b_cont = data_mp;
7810                }
7811                table = (ip6_asp_t *)data_mp->b_rptr;
7812                table_size = iocp->ioc_count;
7813        }

7815        switch (iocp->ioc_cmd) {
7816        case SIOCGIP6ADDRPOLICY:
7817                iocp->ioc_rval = ip6_asp_get(table, table_size, ipst);
7818                if (iocp->ioc_rval == -1)
7819                        iocp->ioc_error = EINVAL;
7820 #if defined(_SYSCALL32_IMPL) && _LONG_LONG_ALIGNMENT_32 == 4
7821                else if (table != NULL &&
7822                    (iocp->ioc_flag & IOC_MODELS) == IOC_ILP32) {
7823                        ip6_asp_t *src = table;
7824                        ip6_asp32_t *dst = (void *)table;
7825                        int count = table_size / sizeof (ip6_asp_t);
7826                        int i;

7828                        /*
7829                         * We need to do an in-place shrink of the array
7830                         * to match the alignment attributes of the
7831                         * 32-bit ABI looking at it.
7832                         */
7833                        /* LINTED: logical expression always true: op "||" */
7834                        ASSERT(sizeof (*src) > sizeof (*dst));
7835                        for (i = 1; i < count; i++)
7836                                bcopy(src + i, dst + i, sizeof (*dst));
7837                }
7838 #endif
7839                break;

7841        case SIOCSIP6ADDRPOLICY:
7842                ASSERT(mp->b_prev == NULL);
7843                mp->b_prev = (void *)q;
7844 #if defined(_SYSCALL32_IMPL) && _LONG_LONG_ALIGNMENT_32 == 4
7845                /*
7846                 * We pass in the datamodel here so that the ip6_asp_replace()
7847                 * routine can handle converting from 32-bit to native formats
7848                 * where necessary.
7849                 *
```



```
7850                 * A better way to handle this might be to convert the inbound
7851                 * data structure here, and hang it off a new 'mp'; thus the
7852                 * ip6_asp_replace() logic would always be dealing with native
7853                 * format data structures..
7854                 *
7855                 * (An even simpler way to handle these ioctls is to just
7856                 * add a 32-bit trailing 'pad' field to the ip6_asp_t structure
7857                 * and just recompile everything that depends on it.)
7858                 */
7859 #endif
7860                ip6_asp_replace(mp, table, table_size, B_FALSE, ipst,
7861                    iocp->ioc_flag & IOC_MODELS);
7862                return;
7863        }

7865        DB_TYPE(mp) =  (iocp->ioc_error == 0) ? M_IOCACK : M_IOCNAK;
7866        qreply(q, mp);
7867 }

7869 static void
7870 ip_sioctl_dstinfo(queue_t *q, mblk_t *mp)
7871 {
7872        mblk_t          *data_mp;
7873        struct dstinforeq       *dir;
7874        uint8_t         *end, *cur;
7875        in6_addr_t      *daddr, *saddr;
7876        ipaddr_t        v4daddr;
7877        ire_t           *ire;
7878        ipaddr_t        v4setsrc;
7879        in6_addr_t      v6setsrc;
7880        char            *slabel, *dlabel;
7881        boolean_t       isipv4;
7882        int             match_ire;
7883        ill_t           *dst_ill;
7884        struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
7885        conn_t          *connp = Q_TO_CONN(q);
7886        zoneid_t        zoneid = IPCL_ZONEID(connp);
7887        ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
7888        uint64_t        ipif_flags;

7890        ASSERT(q->q_next == NULL); /* this ioctl not allowed if ip is module */

7892        /*
7893         * This ioctl is I_STR only, and must have a
7894         * data mblk following the M_IOCTL mblk.
7895         */
7896        data_mp = mp->b_cont;
7897        if (iocp->ioc_count == TRANSPARENT || data_mp == NULL) {
7898                miocnak(q, mp, 0, EINVAL);
7899                return;
7900        }

7902        if (MBLKL(data_mp) < iocp->ioc_count) {
7903                mblk_t *new_data_mp;

7905                if ((new_data_mp = msgpullup(data_mp, -1)) == NULL) {
7906                        miocnak(q, mp, 0, ENOMEM);
7907                        return;
7908                }
7909                freemsg(data_mp);
7910                data_mp = new_data_mp;
7911                mp->b_cont = data_mp;
7912        }
7913        match_ire = MATCH_IRE_DSTONLY;

7915        for (cur = data_mp->b_rptr, end = data_mp->b_wptr;
```

```
7916            end - cur >= sizeof (struct dstinforeq);
7917            cur += sizeof (struct dstinforeq)) {
7918                dir = (struct dstinforeq *)cur;
7919                daddr = &dir->dir_daddr;
7920                saddr = &dir->dir_saddr;

7922                /*
7923                 * ip_addr_scope_v6() and ip6_asp_lookup() handle
7924                 * v4 mapped addresses; ire_ftable_lookup_v6()
7925                 * and ip_select_source_v6() do not.
7926                 */
7927                dir->dir_dscope = ip_addr_scope_v6(daddr);
7928                dlabel = ip6_asp_lookup(daddr, &dir->dir_precedence, ipst);

7930                isipv4 = IN6_IS_ADDR_V4MAPPED(daddr);
7931                if (isipv4) {
7932                        IN6_V4MAPPED_TO_IPADDR(daddr, v4daddr);
7933                        v4setsrc = INADDR_ANY;
7934                        ire = ire_route_recursive_v4(v4daddr, 0, NULL, zoneid,
7935                            NULL, match_ire, IRR_ALLOCATE, 0, ipst, &v4setsrc,
7936                            NULL, NULL);
7937                } else {
7938                        v6setsrc = ipv6_all_zeros;
7939                        ire = ire_route_recursive_v6(daddr, 0, NULL, zoneid,
7940                            NULL, match_ire, IRR_ALLOCATE, 0, ipst, &v6setsrc,
7941                            NULL, NULL);
7942                }
7943                ASSERT(ire != NULL);
7944                if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) {
7945                        ire_refrele(ire);
7946                        dir->dir_dreachable = 0;

7948                        /* move on to next dst addr */
7949                        continue;
7950                }
7951                dir->dir_dreachable = 1;

7953                dst_ill = ire_nexthop_ill(ire);
7954                if (dst_ill == NULL) {
7955                        ire_refrele(ire);
7956                        continue;
7957                }

7959                /* With ipmp we most likely look at the ipmp ill here */
7960                dir->dir_dmactype = dst_ill->ill_mactype;

7962                if (isipv4) {
7963                        ipaddr_t v4saddr;

7965                        if (ip_select_source_v4(dst_ill, v4setsrc, v4daddr,
7966                            connp->conn_ixa->ixa_multicast_ifaddr, zoneid, ipst,
7967                            &v4saddr, NULL, &ipif_flags) != 0) {
7968                                v4saddr = INADDR_ANY;
7969                                ipif_flags = 0;
7970                        }
7971                        IN6_IPADDR_TO_V4MAPPED(v4saddr, saddr);
7972                } else {
7973                        if (ip_select_source_v6(dst_ill, &v6setsrc, daddr,
7974                            zoneid, ipst, B_FALSE, IPV6_PREFER_SRC_DEFAULT,
7975                            saddr, NULL, &ipif_flags) != 0) {
7976                                *saddr = ipv6_all_zeros;
7977                                ipif_flags = 0;
7978                        }
7979                }

7981                dir->dir_sscope = ip_addr_scope_v6(saddr);
```



```
7982                slabel = ip6_asp_lookup(saddr, NULL, ipst);
7983                dir->dir_labelmatch = ip6_asp_labelcmp(dlabel, slabel);
7984                dir->dir_sdeprecated = (ipif_flags & IPIF_DEPRECATED) ? 1 : 0;
7985                ire_refrele(ire);
7986                ill_refrele(dst_ill);
7987        }
7988        miocack(q, mp, iocp->ioc_count, 0);
7989 }

7991 /*
7992  * Check if this is an address assigned to this machine.
7993  * Skips interfaces that are down by using ire checks.
7994  * Translates mapped addresses to v4 addresses and then
7995  * treats them as such, returning true if the v4 address
7996  * associated with this mapped address is configured.
7997  * Note: Applications will have to be careful what they do
7998  * with the response; use of mapped addresses limits
7999  * what can be done with the socket, especially with
8000  * respect to socket options and ioctls - neither IPv4
8001  * options nor IPv6 sticky options/ancillary data options
8002  * may be used.
8003  */
8004 /* ARGSUSED */
8005 int
8006 ip_sioctl_tmyaddr(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
8007     ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
8008 {
8009        struct sioc_addrreq *sia;
8010        sin_t *sin;
8011        ire_t *ire;
8012        mblk_t *mp1;
8013        zoneid_t zoneid;
8014        ip_stack_t      *ipst;

8016        ip1dbg(("ip_sioctl_tmyaddr"));

8018        ASSERT(q->q_next == NULL); /* this ioctl not allowed if ip is module */
8019        zoneid = Q_TO_CONN(q)->conn_zoneid;
8020        ipst = CONNQ_TO_IPST(q);

8022        /* Existence verified in ip_wput_nondata */
8023        mp1 = mp->b_cont->b_cont;
8024        sia = (struct sioc_addrreq *)mp1->b_rptr;
8025        sin = (sin_t *)&sia->sa_addr;
8026        switch (sin->sin_family) {
8027        case AF_INET6: {
8028                sin6_t *sin6 = (sin6_t *)sin;

8030                if (IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
8031                        ipaddr_t v4_addr;

8033                        IN6_V4MAPPED_TO_IPADDR(&sin6->sin6_addr,
8034                            v4_addr);
8035                        ire = ire_ftable_lookup_v4(v4_addr, 0, 0,
8036                            IRE_LOCAL|IRE_LOOPBACK, NULL, zoneid, NULL,
8037                            MATCH_IRE_TYPE | MATCH_IRE_ZONEONLY, 0, ipst, NULL);
8038                } else {
8039                        in6_addr_t v6addr;

8041                        v6addr = sin6->sin6_addr;
8042                        ire = ire_ftable_lookup_v6(&v6addr, 0, 0,
8043                            IRE_LOCAL|IRE_LOOPBACK, NULL, zoneid, NULL,
8044                            MATCH_IRE_TYPE | MATCH_IRE_ZONEONLY, 0, ipst, NULL);
8045                }
8046                break;
8047        }
```

```
8048         case AF_INET: {
8049                 ipaddr_t v4addr;

8051                 v4addr = sin->sin_addr.s_addr;
8052                 ire = ire_ftable_lookup_v4(v4addr, 0, 0,
8053                     IRE_LOCAL|IRE_LOOPBACK, NULL, zoneid,
8054                     NULL, MATCH_IRE_TYPE | MATCH_IRE_ZONEONLY, 0, ipst, NULL);
8055                 break;
8056         }
8057         default:
8058                 return (EAFNOSUPPORT);
8059         }
8060         if (ire != NULL) {
8061                 sia->sa_res = 1;
8062                 ire_refrele(ire);
8063         } else {
8064                 sia->sa_res = 0;
8065         }
8066         return (0);
8067 }

8069 /*
8070  * Check if this is an address assigned on-link i.e. neighbor,
8071  * and makes sure it's reachable from the current zone.
8072  * Returns true for my addresses as well.
8073  * Translates mapped addresses to v4 addresses and then
8074  * treats them as such, returning true if the v4 address
8075  * associated with this mapped address is configured.
8076  * Note: Applications will have to be careful what they do
8077  * with the response; use of mapped addresses limits
8078  * what can be done with the socket, especially with
8079  * respect to socket options and ioctls - neither IPv4
8080  * options nor IPv6 sticky options/ancillary data options
8081  * may be used.
8082  */
8083 /* ARGSUSED */
8084 int
8085 ip_sioctl_tonlink(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
8086     ip_ioctl_cmd_t *ipip, void *duymmy_ifreq)
8087 {
8088         struct sioc_addrreq *sia;
8089         sin_t *sin;
8090         mblk_t  *mp1;
8091         ire_t *ire = NULL;
8092         zoneid_t zoneid;
8093         ip_stack_t      *ipst;

8095         ip1dbg(("ip_sioctl_tonlink"));

8097         ASSERT(q->q_next == NULL); /* this ioctl not allowed if ip is module */
8098         zoneid = Q_TO_CONN(q)->conn_zoneid;
8099         ipst = CONNQ_TO_IPST(q);

8101         /* Existence verified in ip_wput_nondata */
8102         mp1 = mp->b_cont->b_cont;
8103         sia = (struct sioc_addrreq *)mp1->b_rptr;
8104         sin = (sin_t *)&sia->sa_addr;

8106         /*
8107          * We check for IRE_ONLINK and exclude IRE_BROADCAST|IRE_MULTICAST
8108          * to make sure we only look at on-link unicast address.
8109          */
8110         switch (sin->sin_family) {
8111         case AF_INET6: {
8112                 sin6_t *sin6 = (sin6_t *)sin;
```



```
8114                 if (IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
8115                         ipaddr_t v4_addr;

8117                         IN6_V4MAPPED_TO_IPADDR(&sin6->sin6_addr,
8118                             v4_addr);
8119                         if (!CLASSD(v4_addr)) {
8120                                 ire = ire_ftable_lookup_v4(v4_addr, 0, 0, 0,
8121                                     NULL, zoneid, NULL, MATCH_IRE_DSTONLY,
8122                                     0, ipst, NULL);
8123                         }
8124                 } else {
8125                         in6_addr_t v6addr;

8127                         v6addr = sin6->sin6_addr;
8128                         if (!IN6_IS_ADDR_MULTICAST(&v6addr)) {
8129                                 ire = ire_ftable_lookup_v6(&v6addr, 0, 0, 0,
8130                                     NULL, zoneid, NULL, MATCH_IRE_DSTONLY, 0,
8131                                     ipst, NULL);
8132                         }
8133                 }
8134                 break;
8135         }
8136         case AF_INET: {
8137                 ipaddr_t v4addr;

8139                 v4addr = sin->sin_addr.s_addr;
8140                 if (!CLASSD(v4addr)) {
8141                         ire = ire_ftable_lookup_v4(v4addr, 0, 0, 0, NULL,
8142                             zoneid, NULL, MATCH_IRE_DSTONLY, 0, ipst, NULL);
8143                 }
8144                 break;
8145         }
8146         default:
8147                 return (EAFNOSUPPORT);
8148         }
8149         sia->sa_res = 0;
8150         if (ire != NULL) {
8151                 ASSERT(!(ire->ire_type & IRE_MULTICAST));

8153                 if ((ire->ire_type & IRE_ONLINK) &&
8154                     !(ire->ire_type & IRE_BROADCAST))
8155                         sia->sa_res = 1;
8156                 ire_refrele(ire);
8157         }
8158         return (0);
8159 }

8161 /*
8162  * TBD: implement when kernel maintaines a list of site prefixes.
8163  */
8164 /* ARGSUSED */
8165 int
8166 ip_sioctl_tmysite(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
8167     ip_ioctl_cmd_t *ipip, void *ifreq)
8168 {
8169         return (ENXIO);
8170 }

8172 /* ARP IOCTLs. */
8173 /* ARGSUSED */
8174 int
8175 ip_sioctl_arp(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
8176     ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
8177 {
8178         int             err;
8179         ipaddr_t        ipaddr;
```

```
8180         struct iocblk   *iocp;
8181         conn_t          *connp;
8182         struct arpreq   *ar;
8183         struct xarpreq  *xar;
8184         int             arp_flags, flags, alength;
8185         uchar_t         *lladdr;
8186         ip_stack_t      *ipst;
8187         ill_t           *ill = ipif->ipif_ill;
8188         ill_t           *proxy_ill = NULL;
8189         ipmp_arpent_t   *entp = NULL;
8190         boolean_t       proxyarp = B_FALSE;
8191         boolean_t       if_arp_ioctl = B_FALSE;
8192         ncec_t          *ncec = NULL;
8193         nce_t           *nce;

8195         ASSERT(!(q->q_flag & QREADR) && q->q_next == NULL);
8196         connp = Q_TO_CONN(q);
8197         ipst = connp->conn_netstack->netstack_ip;
8198         iocp = (struct iocblk *)mp->b_rptr;

8200         if (ipip->ipi_cmd_type == XARP_CMD) {
8201                 /* We have a chain - M_IOCTL-->MI_COPY_MBLK-->XARPREQ_MBLK */
8202                 xar = (struct xarpreq *)mp->b_cont->b_cont->b_rptr;
8203                 ar = NULL;

8205                 arp_flags = xar->xarp_flags;
8206                 lladdr = (uchar_t *)LLADDR(&xar->xarp_ha);
8207                 if_arp_ioctl = (xar->xarp_ha.sdl_nlen != 0);
8208                 /*
8209                  * Validate against user's link layer address length
8210                  * input and name and addr length limits.
8211                  */
8212                 alength = ill->ill_phys_addr_length;
8213                 if (ipip->ipi_cmd == SIOCSXARP) {
8214                         if (alength != xar->xarp_ha.sdl_alen ||
8215                             (alength + xar->xarp_ha.sdl_nlen >
8216                             sizeof (xar->xarp_ha.sdl_data)))
8217                                 return (EINVAL);
8218                 }
8219         } else {
8220                 /* We have a chain - M_IOCTL-->MI_COPY_MBLK-->ARPREQ_MBLK */
8221                 ar = (struct arpreq *)mp->b_cont->b_cont->b_rptr;
8222                 xar = NULL;

8224                 arp_flags = ar->arp_flags;
8225                 lladdr = (uchar_t *)ar->arp_ha.sa_data;
8226                 /*
8227                  * Theoretically, the sa_family could tell us what link
8228                  * layer type this operation is trying to deal with. By
8229                  * common usage AF_UNSPEC means ethernet. We'll assume
8230                  * any attempt to use the SIOC?ARP ioctls is for ethernet,
8231                  * for now. Our new SIOC*XARP ioctls can be used more
8232                  * generally.
8233                  *
8234                  * If the underlying media happens to have a non 6 byte
8235                  * address, arp module will fail set/get, but the del
8236                  * operation will succeed.
8237                  */
8238                 alength = 6;
8239                 if ((ipip->ipi_cmd != SIOCDARP) &&
8240                     (alength != ill->ill_phys_addr_length)) {
8241                         return (EINVAL);
8242                 }
8243         }

8245         /* Translate ATF* flags to NCE* flags */
```



```
8246         flags = 0;
8247         if (arp_flags & ATF_AUTHORITY)
8248                 flags |= NCE_F_AUTHORITY;
8249         if (arp_flags & ATF_PERM)
8250                 flags |= NCE_F_NONUD; /* not subject to aging */
8251         if (arp_flags & ATF_PUBL)
8252                 flags |= NCE_F_PUBLISH;

8254         /*
8255          * IPMP ARP special handling:
8256          *
8257          * 1. Since ARP mappings must appear consistent across the group,
8258          *    prohibit changing ARP mappings on the underlying interfaces.
8259          *
8260          * 2. Since ARP mappings for IPMP data addresses are maintained by
8261          *    IP itself, prohibit changing them.
8262          *
8263          * 3. For proxy ARP, use a functioning hardware address in the group,
8264          *    provided one exists.  If one doesn't, just add the entry as-is;
8265          *    ipmp_illgrp_refresh_arpent() will refresh it if things change.
8266          */
8267         if (IS_UNDER_IPMP(ill)) {
8268                 if (ipip->ipi_cmd != SIOCGARP && ipip->ipi_cmd != SIOCGXARP)
8269                         return (EPERM);
8270         }
8271         if (IS_IPMP(ill)) {
8272                 ipmp_illgrp_t *illg = ill->ill_grp;

8274                 switch (ipip->ipi_cmd) {
8275                 case SIOCSARP:
8276                 case SIOCSXARP:
8277                         proxy_ill = ipmp_illgrp_find_ill(illg, lladdr, alength);
8278                         if (proxy_ill != NULL) {
8279                                 proxyarp = B_TRUE;
8280                                 if (!ipmp_ill_is_active(proxy_ill))
8281                                         proxy_ill = ipmp_illgrp_next_ill(illg);
8282                                 if (proxy_ill != NULL)
8283                                         lladdr = proxy_ill->ill_phys_addr;
8284                         }
8285                         /* FALLTHRU */
8286                 }
8287         }

8289         ipaddr = sin->sin_addr.s_addr;
8290         /*
8291          * don't match across illgrp per case (1) and (2).
8292          * XXX use IS_IPMP(ill) like ndp_sioc_update?
8293          */
8294         nce = nce_lookup_v4(ill, &ipaddr);
8295         if (nce != NULL)
8296                 ncec = nce->nce_common;

8298         switch (iocp->ioc_cmd) {
8299         case SIOCDARP:
8300         case SIOCDXARP: {
8301                 /*
8302                  * Delete the NCE if any.
8303                  */
8304                 if (ncec == NULL) {
8305                         iocp->ioc_error = ENXIO;
8306                         break;
8307                 }
8308                 /* Don't allow changes to arp mappings of local addresses. */
8309                 if (NCE_MYADDR(ncec)) {
8310                         nce_refrele(nce);
8311                         return (ENOTSUP);
```

```
8312                }
8313                iocp->ioc_error = 0;

8315                /*
8316                 * Delete the nce_common which has ncec_ill set to ipmp_ill.
8317                 * This will delete all the nce entries on the under_ills.
8318                 */
8319                ncec_delete(ncec);
8320                /*
8321                 * Once the NCE has been deleted, then the ire_dep* consistency
8322                 * mechanism will find any IRE which depended on the now
8323                 * condemned NCE (as part of sending packets).
8324                 * That mechanism handles redirects by deleting redirects
8325                 * that refer to UNREACHABLE nces.
8326                 */
8327                break;
8328        }
8329        case SIOCGARP:
8330        case SIOCGXARP:
8331                if (ncec != NULL) {
8332                        lladdr = ncec->ncec_lladdr;
8333                        flags = ncec->ncec_flags;
8334                        iocp->ioc_error = 0;
8335                        ip_sioctl_garp_reply(mp, ncec->ncec_ill, lladdr, flags);
8336                } else {
8337                        iocp->ioc_error = ENXIO;
8338                }
8339                break;
8340        case SIOCSARP:
8341        case SIOCSXARP:
8342                /* Don't allow changes to arp mappings of local addresses. */
8343                if (ncec != NULL && NCE_MYADDR(ncec)) {
8344                        nce_refrele(nce);
8345                        return (ENOTSUP);
8346                }

8348                /* static arp entries will undergo NUD if ATF_PERM is not set */
8349                flags |= NCE_F_STATIC;
8350                if (!if_arp_ioctl) {
8351                        ip_nce_lookup_and_update(&ipaddr, NULL, ipst,
8352                            lladdr, alength, flags);
8353                } else {
8354                        ipif_t *ipif = ipif_get_next_ipif(NULL, ill);
8355                        if (ipif != NULL) {
8356                                ip_nce_lookup_and_update(&ipaddr, ipif, ipst,
8357                                    lladdr, alength, flags);
8358                                ipif_refrele(ipif);
8359                        }
8360                }
8361                if (nce != NULL) {
8362                        nce_refrele(nce);
8363                        nce = NULL;
8364                }
8365                /*
8366                 * NCE_F_STATIC entries will be added in state ND_REACHABLE
8367                 * by nce_add_common()
8368                 */
8369                err = nce_lookup_then_add_v4(ill, lladdr,
8370                    ill->ill_phys_addr_length, &ipaddr, flags, ND_UNCHANGED,
8371                    &nce);
8372                if (err == EEXIST) {
8373                        ncec = nce->nce_common;
8374                        mutex_enter(&ncec->ncec_lock);
8375                        ncec->ncec_state = ND_REACHABLE;
8376                        ncec->ncec_flags = flags;
8377                        nce_update(ncec, ND_UNCHANGED, lladdr);
```



```
8378                        mutex_exit(&ncec->ncec_lock);
8379                        err = 0;
8380                }
8381                if (nce != NULL) {
8382                        nce_refrele(nce);
8383                        nce = NULL;
8384                }
8385                if (IS_IPMP(ill) && err == 0) {
8386                        entp = ipmp_illgrp_create_arpent(ill->ill_grp,
8387                            proxyarp, ipaddr, lladdr, ill->ill_phys_addr_length,
8388                            flags);
8389                        if (entp == NULL || (proxyarp && proxy_ill == NULL)) {
8390                                iocp->ioc_error = (entp == NULL ? ENOMEM : 0);
8391                                break;
8392                        }
8393                }
8394                iocp->ioc_error = err;
8395        }

8397        if (nce != NULL) {
8398                nce_refrele(nce);
8399        }

8401        /*
8402         * If we created an IPMP ARP entry, mark that we've notified ARP.
8403         */
8404        if (entp != NULL)
8405                ipmp_illgrp_mark_arpent(ill->ill_grp, entp);

8407        return (iocp->ioc_error);
8408 }

8410 /*
8411  * Parse an [x]arpreq structure coming down SIOC[GSD][X]ARP ioctls, identify
8412  * the associated sin and refhold and return the associated ipif via 'ci'.
8413  */
8414 int
8415 ip_extract_arpreq(queue_t *q, mblk_t *mp, const ip_ioctl_cmd_t *ipip,
8416     cmd_info_t *ci)
8417 {
8418        mblk_t  *mp1;
8419        sin_t   *sin;
8420        conn_t  *connp;
8421        ipif_t  *ipif;
8422        ire_t   *ire = NULL;
8423        ill_t   *ill = NULL;
8424        boolean_t exists;
8425        ip_stack_t *ipst;
8426        struct arpreq *ar;
8427        struct xarpreq *xar;
8428        struct sockaddr_dl *sdl;

8430        /* ioctl comes down on a conn */
8431        ASSERT(!(q->q_flag & QREADR) && q->q_next == NULL);
8432        connp = Q_TO_CONN(q);
8433        if (connp->conn_family == AF_INET6)
8434                return (ENXIO);

8436        ipst = connp->conn_netstack->netstack_ip;

8438        /* Verified in ip_wput_nondata */
8439        mp1 = mp->b_cont->b_cont;

8441        if (ipip->ipi_cmd_type == XARP_CMD) {
8442                ASSERT(MBLKL(mp1) >= sizeof (struct xarpreq));
8443                xar = (struct xarpreq *)mp1->b_rptr;
```

```
8444                sin = (sin_t *)&xar->xarp_pa;
8445                sdl = &xar->xarp_ha;

8447                if (sdl->sdl_family != AF_LINK || sin->sin_family != AF_INET)
8448                        return (ENXIO);
8449                if (sdl->sdl_nlen >= LIFNAMSIZ)
8450                        return (EINVAL);
8451        } else {
8452                ASSERT(ipip->ipi_cmd_type == ARP_CMD);
8453                ASSERT(MBLKL(mp1) >= sizeof (struct arpreq));
8454                ar = (struct arpreq *)mp1->b_rptr;
8455                sin = (sin_t *)&ar->arp_pa;
8456        }

8458        if (ipip->ipi_cmd_type == XARP_CMD && sdl->sdl_nlen != 0) {
8459                ipif = ipif_lookup_on_name(sdl->sdl_data, sdl->sdl_nlen,
8460                    B_FALSE, &exists, B_FALSE, ALL_ZONES, ipst);
8461                if (ipif == NULL)
8462                        return (ENXIO);
8463                if (ipif->ipif_id != 0) {
8464                        ipif_refrele(ipif);
8465                        return (ENXIO);
8466                }
8467        } else {
8468                /*
8469                 * Either an SIOC[DGS]ARP or an SIOC[DGS]XARP with an sdl_nlen
8470                 * of 0: use the IP address to find the ipif.  If the IP
8471                 * address is an IPMP test address, ire_ftable_lookup() will
8472                 * find the wrong ill, so we first do an ipif_lookup_addr().
8473                 */
8474                ipif = ipif_lookup_addr(sin->sin_addr.s_addr, NULL, ALL_ZONES,
8475                    ipst);
8476                if (ipif == NULL) {
8477                        ire = ire_ftable_lookup_v4(sin->sin_addr.s_addr,
8478                            0, 0, IRE_IF_RESOLVER, NULL, ALL_ZONES,
8479                            NULL, MATCH_IRE_TYPE, 0, ipst, NULL);
8480                        if (ire == NULL || ((ill = ire->ire_ill) == NULL)) {
8481                                if (ire != NULL)
8482                                        ire_refrele(ire);
8483                                return (ENXIO);
8484                        }
8485                        ASSERT(ire != NULL && ill != NULL);
8486                        ipif = ill->ill_ipif;
8487                        ipif_refhold(ipif);
8488                        ire_refrele(ire);
8489                }
8490        }

8492        if (ipif->ipif_ill->ill_net_type != IRE_IF_RESOLVER) {
8493                ipif_refrele(ipif);
8494                return (ENXIO);
8495        }

8497        ci->ci_sin = sin;
8498        ci->ci_ipif = ipif;
8499        return (0);
8500 }

8502 /*
8503  * Link or unlink the illgrp on IPMP meta-interface 'ill' depending on the
8504  * value of 'ioccmd'.  While an illgrp is linked to an ipmp_grp_t, it is
8505  * accessible from that ipmp_grp_t, which means SIOCSLIFGROUPNAME can look it
8506  * up and thus an ill can join that illgrp.
8507  *
8508  * We use I_PLINK/I_PUNLINK to do the link/unlink operations rather than
8509  * open()/close() primarily because close() is not allowed to fail or block
```



```
8510  * forever.  On the other hand, I_PUNLINK *can* fail, and there's no reason
8511  * why anyone should ever need to I_PUNLINK an in-use IPMP stream.  To ensure
8512  * symmetric behavior (e.g., doing an I_PLINK after and I_PUNLINK undoes the
8513  * I_PUNLINK) we defer linking to I_PLINK.  Separately, we also fail attempts
8514  * to I_LINK since I_UNLINK is optional and we'd end up in an inconsistent
8515  * state if I_UNLINK didn't occur.
8516  *
8517  * Note that for each plumb/unplumb operation, we may end up here more than
8518  * once because of the way ifconfig works.  However, it's OK to link the same
8519  * illgrp more than once, or unlink an illgrp that's already unlinked.
8520  */
8521 static int
8522 ip_sioctl_plink_ipmp(ill_t *ill, int ioccmd)
8523 {
8524         int err;
8525         ip_stack_t *ipst = ill->ill_ipst;

8527         ASSERT(IS_IPMP(ill));
8528         ASSERT(IAM_WRITER_ILL(ill));

8530         switch (ioccmd) {
8531         case I_LINK:
8532                 return (ENOTSUP);

8534         case I_PLINK:
8535                 rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);
8536                 ipmp_illgrp_link_grp(ill->ill_grp, ill->ill_phyint->phyint_grp);
8537                 rw_exit(&ipst->ips_ipmp_lock);
8538                 break;

8540         case I_PUNLINK:
8541                 /*
8542                  * Require all UP ipifs be brought down prior to unlinking the
8543                  * illgrp so any associated IREs (and other state) is torched.
8544                  */
8545                 if (ill->ill_ipif_up_count + ill->ill_ipif_dup_count > 0)
8546                         return (EBUSY);

8548                 /*
8549                  * NOTE: We hold ipmp_lock across the unlink to prevent a race
8550                  * with an SIOCSLIFGROUPNAME request from an ill trying to
8551                  * join this group.  Specifically: ills trying to join grab
8552                  * ipmp_lock and bump a "pending join" counter checked by
8553                  * ipmp_illgrp_unlink_grp().  During the unlink no new pending
8554                  * joins can occur (since we have ipmp_lock).  Once we drop
8555                  * ipmp_lock, subsequent SIOCSLIFGROUPNAME requests will not
8556                  * find the illgrp (since we unlinked it) and will return
8557                  * EAFNOSUPPORT.  This will then take them back through the
8558                  * IPMP meta-interface plumbing logic in ifconfig, and thus
8559                  * back through I_PLINK above.
8560                  */
8561                 rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);
8562                 err = ipmp_illgrp_unlink_grp(ill->ill_grp);
8563                 rw_exit(&ipst->ips_ipmp_lock);
8564                 return (err);
8565         default:
8566                 break;
8567         }
8568         return (0);
8569 }

8571 /*
8572  * Do I_PLINK/I_LINK or I_PUNLINK/I_UNLINK with consistency checks and also
8573  * atomically set/clear the muxids. Also complete the ioctl by acking or
8574  * naking it.  Note that the code is structured such that the link type,
8575  * whether it's persistent or not, is treated equally.  ifconfig(1M) and
```

```
 * its clones use the persistent link, while pppd(1M) and perhaps many
 * other daemons may use non-persistent link.  When combined with some
 * ill_t states, linking and unlinking lower streams may be used as
 * indicators of dynamic re-plumbing events [see PSARC/1999/348].
 */
/* ARGSUSED */
void
ip_sioctl_plink(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
{
        mblk_t          *mp1;
        struct linkblk  *li;
        int             ioccmd = ((struct iocblk *)mp->b_rptr)->ioc_cmd;
        int             err = 0;

        ASSERT(ioccmd == I_PLINK || ioccmd == I_PUNLINK ||
            ioccmd == I_LINK || ioccmd == I_UNLINK);

        mp1 = mp->b_cont;       /* This is the linkblk info */
        li = (struct linkblk *)mp1->b_rptr;

        err = ip_sioctl_plink_ipmod(ipsq, q, mp, ioccmd, li);
        if (err == EINPROGRESS)
                return;
        if (err == 0)
                miocack(q, mp, 0, 0);
        else
                miocnak(q, mp, 0, err);

        /* Conn was refheld in ip_sioctl_copyin_setup */
        if (CONN_Q(q)) {
                CONN_DEC_IOCTLREF(Q_TO_CONN(q));
                CONN_OPER_PENDING_DONE(Q_TO_CONN(q));
        }
}

/*
 * Process I_{P}LINK and I_{P}UNLINK requests named by 'ioccmd' and pointed to
 * by 'mp' and 'li' for the IP module stream (if li->q_bot is in fact an IP
 * module stream).
 * Returns zero on success, EINPROGRESS if the operation is still pending, or
 * an error code on failure.
 */
static int
ip_sioctl_plink_ipmod(ipsq_t *ipsq, queue_t *q, mblk_t *mp, int ioccmd,
    struct linkblk *li)
{
        int             err = 0;
        ill_t           *ill;
        queue_t         *ipwq, *dwq;
        const char      *name;
        struct qinit    *qinfo;
        boolean_t       islink = (ioccmd == I_PLINK || ioccmd == I_LINK);
        boolean_t       entered_ipsq = B_FALSE;
        boolean_t       is_ip = B_FALSE;
        arl_t           *arl;

        /*
         * Walk the lower stream to verify it's the IP module stream.
         * The IP module is identified by its name, wput function,
         * and non-NULL q_next.  STREAMS ensures that the lower stream
         * (li->l_qbot) will not vanish until this ioctl completes.
         */
        for (ipwq = li->l_qbot; ipwq != NULL; ipwq = ipwq->q_next) {
                qinfo = ipwq->q_qinfo;
                name = qinfo->qi_minfo->mi_idname;
                if (name != NULL && strcmp(name, ip_mod_info.mi_idname) == 0 &&
```



```
                    qinfo->qi_putp != (pfi_t)ip_lwput && ipwq->q_next != NULL) {
                        is_ip = B_TRUE;
                        break;
                }
                if (name != NULL && strcmp(name, arp_mod_info.mi_idname) == 0 &&
                    qinfo->qi_putp != (pfi_t)ip_lwput && ipwq->q_next != NULL) {
                        break;
                }
        }

        /*
         * If this isn't an IP module stream, bail.
         */
        if (ipwq == NULL)
                return (0);

        if (!is_ip) {
                arl = (arl_t *)ipwq->q_ptr;
                ill = arl_to_ill(arl);
                if (ill == NULL)
                        return (0);
        } else {
                ill = ipwq->q_ptr;
        }
        ASSERT(ill != NULL);

        if (ipsq == NULL) {
                ipsq = ipsq_try_enter(NULL, ill, q, mp, ip_sioctl_plink,
                    NEW_OP, B_FALSE);
                if (ipsq == NULL) {
                        if (!is_ip)
                                ill_refrele(ill);
                        return (EINPROGRESS);
                }
                entered_ipsq = B_TRUE;
        }
        ASSERT(IAM_WRITER_ILL(ill));
        mutex_enter(&ill->ill_lock);
        if (!is_ip) {
                if (islink && ill->ill_muxid == 0) {
                        /*
                         * Plumbing has to be done with IP plumbed first, arp
                         * second, but here we have arp being plumbed first.
                         */
                        mutex_exit(&ill->ill_lock);
                        if (entered_ipsq)
                                ipsq_exit(ipsq);
                        ill_refrele(ill);
                        return (EINVAL);
                }
        }
        mutex_exit(&ill->ill_lock);
        if (!is_ip) {
                arl->arl_muxid = islink ? li->l_index : 0;
                ill_refrele(ill);
                goto done;
        }

        if (IS_IPMP(ill) && (err = ip_sioctl_plink_ipmp(ill, ioccmd)) != 0)
                goto done;

        /*
         * As part of I_{P}LINKing, stash the number of downstream modules and
         * the read queue of the module immediately below IP in the ill.
         * These are used during the capability negotiation below.
         */
```

```
8708         ill->ill_lmod_rq = NULL;
8709         ill->ill_lmod_cnt = 0;
8710         if (islink && ((dwq = ipwq->q_next) != NULL)) {
8711                 ill->ill_lmod_rq = RD(dwq);
8712                 for (; dwq != NULL; dwq = dwq->q_next)
8713                         ill->ill_lmod_cnt++;
8714         }

8716         ill->ill_muxid = islink ? li->l_index : 0;

8718         /*
8719          * Mark the ipsq busy until the capability operations initiated below
8720          * complete. The PLINK/UNLINK ioctl itself completes when our caller
8721          * returns, but the capability operation may complete asynchronously
8722          * much later.
8723          */
8724         ipsq_current_start(ipsq, ill->ill_ipif, ioccmd);
8725         /*
8726          * If there's at least one up ipif on this ill, then we're bound to
8727          * the underlying driver via DLPI.  In that case, renegotiate
8728          * capabilities to account for any possible change in modules
8729          * interposed between IP and the driver.
8730          */
8731         if (ill->ill_ipif_up_count > 0) {
8732                 if (islink)
8733                         ill_capability_probe(ill);
8734                 else
8735                         ill_capability_reset(ill, B_FALSE);
8736         }
8737         ipsq_current_finish(ipsq);
8738 done:
8739         if (entered_ipsq)
8740                 ipsq_exit(ipsq);

8742         return (err);
8743 }

8745 /*
8746  * Search the ioctl command in the ioctl tables and return a pointer
8747  * to the ioctl command information. The ioctl command tables are
8748  * static and fully populated at compile time.
8749  */
8750 ip_ioctl_cmd_t *
8751 ip_sioctl_lookup(int ioc_cmd)
8752 {
8753         int index;
8754         ip_ioctl_cmd_t *ipip;
8755         ip_ioctl_cmd_t *ipip_end;

8757         if (ioc_cmd == IPI_DONTCARE)
8758                 return (NULL);

8760         /*
8761          * Do a 2 step search. First search the indexed table
8762          * based on the least significant byte of the ioctl cmd.
8763          * If we don't find a match, then search the misc table
8764          * serially.
8765          */
8766         index = ioc_cmd & 0xFF;
8767         if (index < ip_ndx_ioctl_count) {
8768                 ipip = &ip_ndx_ioctl_table[index];
8769                 if (ipip->ipi_cmd == ioc_cmd) {
8770                         /* Found a match in the ndx table */
8771                         return (ipip);
8772                 }
8773         }
```



```
8775         /* Search the misc table */
8776         ipip_end = &ip_misc_ioctl_table[ip_misc_ioctl_count];
8777         for (ipip = ip_misc_ioctl_table; ipip < ipip_end; ipip++) {
8778                 if (ipip->ipi_cmd == ioc_cmd)
8779                         /* Found a match in the misc table */
8780                         return (ipip);
8781         }

8783         return (NULL);
8784 }

8786 /*
8787  * helper function for ip_sioctl_getsetprop(), which does some sanity checks
8788  */
8789 static boolean_t
8790 getset_ioctl_checks(mblk_t *mp)
8791 {
8792         struct iocblk   *iocp = (struct iocblk *)mp->b_rptr;
8793         mblk_t          *mp1 = mp->b_cont;
8794         mod_ioc_prop_t  *pioc;
8795         uint_t          flags;
8796         uint_t          pioc_size;

8798         /* do sanity checks on various arguments */
8799         if (mp1 == NULL || iocp->ioc_count == 0 ||
8800             iocp->ioc_count == TRANSPARENT) {
8801                 return (B_FALSE);
8802         }
8803         if (msgdsize(mp1) < iocp->ioc_count) {
8804                 if (!pullupmsg(mp1, iocp->ioc_count))
8805                         return (B_FALSE);
8806         }

8808         pioc = (mod_ioc_prop_t *)mp1->b_rptr;

8810         /* sanity checks on mpr_valsize */
8811         pioc_size = sizeof (mod_ioc_prop_t);
8812         if (pioc->mpr_valsize != 0)
8813                 pioc_size += pioc->mpr_valsize - 1;

8815         if (iocp->ioc_count != pioc_size)
8816                 return (B_FALSE);

8818         flags = pioc->mpr_flags;
8819         if (iocp->ioc_cmd == SIOCSETPROP) {
8820                 /*
8821                  * One can either reset the value to it's default value or
8822                  * change the current value or append/remove the value from
8823                  * a multi-valued properties.
8824                  */
8825                 if ((flags & MOD_PROP_DEFAULT) != MOD_PROP_DEFAULT &&
8826                     flags != MOD_PROP_ACTIVE &&
8827                     flags != (MOD_PROP_ACTIVE|MOD_PROP_APPEND) &&
8828                     flags != (MOD_PROP_ACTIVE|MOD_PROP_REMOVE))
8829                         return (B_FALSE);
8830         } else {
8831                 ASSERT(iocp->ioc_cmd == SIOCGETPROP);

8833                 /*
8834                  * One can retrieve only one kind of property information
8835                  * at a time.
8836                  */
8837                 if ((flags & MOD_PROP_ACTIVE) != MOD_PROP_ACTIVE &&
8838                     (flags & MOD_PROP_DEFAULT) != MOD_PROP_DEFAULT &&
8839                     (flags & MOD_PROP_POSSIBLE) != MOD_PROP_POSSIBLE &&
```

```
8840                    (flags & MOD_PROP_PERM) != MOD_PROP_PERM)
8841                        return (B_FALSE);
8842        }

8844        return (B_TRUE);
8845 }

8847 /*
8848  * process the SIOC{SET|GET}PROP ioctl's
8849  */
8850 /* ARGSUSED */
8851 static void
8852 ip_sioctl_getsetprop(queue_t *q, mblk_t *mp)
8853 {
8854        struct iocblk   *iocp = (struct iocblk *)mp->b_rptr;
8855        mblk_t          *mp1 = mp->b_cont;
8856        mod_ioc_prop_t  *pioc;
8857        mod_prop_info_t *ptbl = NULL, *pinfo = NULL;
8858        ip_stack_t      *ipst;
8859        icmp_stack_t    *is;
8860        tcp_stack_t     *tcps;
8861        sctp_stack_t    *sctps;
8862        dccp_stack_t    *dccps;
8863 #endif /* ! codereview */
8864        udp_stack_t     *us;
8865        netstack_t      *stack;
8866        void            *cbarg;
8867        cred_t          *cr;
8868        boolean_t       set;
8869        int             err;

8871        ASSERT(q->q_next == NULL);
8872        ASSERT(CONN_Q(q));

8874        if (!getset_ioctl_checks(mp)) {
8875                miocnak(q, mp, 0, EINVAL);
8876                return;
8877        }
8878        ipst = CONNQ_TO_IPST(q);
8879        stack = ipst->ips_netstack;
8880        pioc = (mod_ioc_prop_t *)mp1->b_rptr;

8882        switch (pioc->mpr_proto) {
8883        case MOD_PROTO_IP:
8884        case MOD_PROTO_IPV4:
8885        case MOD_PROTO_IPV6:
8886                ptbl = ipst->ips_propinfo_tbl;
8887                cbarg = ipst;
8888                break;
8889        case MOD_PROTO_RAWIP:
8890                is = stack->netstack_icmp;
8891                ptbl = is->is_propinfo_tbl;
8892                cbarg = is;
8893                break;
8894        case MOD_PROTO_TCP:
8895                tcps = stack->netstack_tcp;
8896                ptbl = tcps->tcps_propinfo_tbl;
8897                cbarg = tcps;
8898                break;
8899        case MOD_PROTO_UDP:
8900                us = stack->netstack_udp;
8901                ptbl = us->us_propinfo_tbl;
8902                cbarg = us;
8903                break;
8904        case MOD_PROTO_SCTP:
8905                sctps = stack->netstack_sctp;
```



```
8906                ptbl = sctps->sctps_propinfo_tbl;
8907                cbarg = sctps;
8908                break;
8909        case MOD_PROTO_DCCP:
8910                dccps = stack->netstack_dccp;
8911                ptbl = dccps->dccps_propinfo_tbl;
8912                cbarg = dccps;
8913 #endif /* ! codereview */
8914        default:
8915                miocnak(q, mp, 0, EINVAL);
8916                return;
8917        }

8919        /* search for given property in respective protocol propinfo table */
8920        for (pinfo = ptbl; pinfo->mpi_name != NULL; pinfo++) {
8921                if (strcmp(pinfo->mpi_name, pioc->mpr_name) == 0 &&
8922                    pinfo->mpi_proto == pioc->mpr_proto)
8923                        break;
8924        }
8925        if (pinfo->mpi_name == NULL) {
8926                miocnak(q, mp, 0, ENOENT);
8927                return;
8928        }

8930        set = (iocp->ioc_cmd == SIOCSETPROP) ? B_TRUE : B_FALSE;
8931        if (set && pinfo->mpi_setf != NULL) {
8932                cr = msg_getcred(mp, NULL);
8933                if (cr == NULL)
8934                        cr = iocp->ioc_cr;
8935                err = pinfo->mpi_setf(cbarg, cr, pinfo, pioc->mpr_ifname,
8936                    pioc->mpr_val, pioc->mpr_flags);
8937        } else if (!set && pinfo->mpi_getf != NULL) {
8938                err = pinfo->mpi_getf(cbarg, pinfo, pioc->mpr_ifname,
8939                    pioc->mpr_val, pioc->mpr_valsize, pioc->mpr_flags);
8940        } else {
8941                err = EPERM;
8942        }

8944        if (err != 0) {
8945                miocnak(q, mp, 0, err);
8946        } else {
8947                if (set)
8948                        miocack(q, mp, 0, 0);
8949                else    /* For get, we need to return back the data */
8950                        miocack(q, mp, iocp->ioc_count, 0);
8951        }
8952 }

8954 /*
8955  * process the legacy ND_GET, ND_SET ioctl just for {ip|ip6}_forwarding
8956  * as several routing daemons have unfortunately used this 'unpublished'
8957  * but well-known ioctls.
8958  */
8959 /* ARGSUSED */
8960 static void
8961 ip_process_legacy_nddprop(queue_t *q, mblk_t *mp)
8962 {
8963        struct iocblk   *iocp = (struct iocblk *)mp->b_rptr;
8964        mblk_t          *mp1 = mp->b_cont;
8965        char            *pname, *pval, *buf;
8966        uint_t          bufsize, proto;
8967        mod_prop_info_t *ptbl = NULL, *pinfo = NULL;
8968        ip_stack_t      *ipst;
8969        int             err = 0;

8971        ASSERT(CONN_Q(q));
```

```
8972         ipst = CONNQ_TO_IPST(q);

8974         if (iocp->ioc_count == 0 || mp1 == NULL) {
8975                 miocnak(q, mp, 0, EINVAL);
8976                 return;
8977         }

8979         mp1->b_datap->db_lim[-1] = '\0';        /* Force null termination */
8980         pval = buf = pname = (char *)mp1->b_rptr;
8981         bufsize = MBLKL(mp1);

8983         if (strcmp(pname, "ip_forwarding") == 0) {
8984                 pname = "forwarding";
8985                 proto = MOD_PROTO_IPV4;
8986         } else if (strcmp(pname, "ip6_forwarding") == 0) {
8987                 pname = "forwarding";
8988                 proto = MOD_PROTO_IPV6;
8989         } else {
8990                 miocnak(q, mp, 0, EINVAL);
8991                 return;
8992         }

8994         ptbl = ipst->ips_propinfo_tbl;
8995         for (pinfo = ptbl; pinfo->mpi_name != NULL; pinfo++) {
8996                 if (strcmp(pinfo->mpi_name, pname) == 0 &&
8997                     pinfo->mpi_proto == proto)
8998                         break;
8999         }

9001         ASSERT(pinfo->mpi_name != NULL);

9003         switch (iocp->ioc_cmd) {
9004         case ND_GET:
9005                 if ((err = pinfo->mpi_getf(ipst, pinfo, NULL, buf, bufsize,
9006                     0)) == 0) {
9007                         miocack(q, mp, iocp->ioc_count, 0);
9008                         return;
9009                 }
9010                 break;
9011         case ND_SET:
9012                 /*
9013                  * buffer will have property name and value in the following
9014                  * format,
9015                  * <property name>'\0'<property value>'\0', extract them;
9016                  */
9017                 while (*pval++)
9018                         noop;

9020                 if (!*pval || pval >= (char *)mp1->b_wptr) {
9021                         err = EINVAL;
9022                 } else if ((err = pinfo->mpi_setf(ipst, NULL, pinfo, NULL,
9023                     pval, 0)) == 0) {
9024                         miocack(q, mp, 0, 0);
9025                         return;
9026                 }
9027                 break;
9028         default:
9029                 err = EINVAL;
9030                 break;
9031         }
9032         miocnak(q, mp, 0, err);
9033 }

9035 /*
9036  * Wrapper function for resuming deferred ioctl processing
9037  * Used for SIOCGDSTINFO, SIOCGIP6ADDRPOLICY, SIOCGMSFILTER,
```



```
9038  * SIOCSMSFILTER, SIOCGIPMSFILTER, and SIOCSIPMSFILTER currently.
9039  */
9040 /* ARGSUSED */
9041 void
9042 ip_sioctl_copyin_resume(ipsq_t *dummy_ipsq, queue_t *q, mblk_t *mp,
9043     void *dummy_arg)
9044 {
9045         ip_sioctl_copyin_setup(q, mp);
9046 }

9048 /*
9049  * ip_sioctl_copyin_setup is called by ip_wput_nondata with any M_IOCTL message
9050  * that arrives.  Most of the IOCTLs are "socket" IOCTLs which we handle
9051  * in either I_STR or TRANSPARENT form, using the mi_copy facility.
9052  * We establish here the size of the block to be copied in.  mi_copyin
9053  * arranges for this to happen, an processing continues in ip_wput_nondata with
9054  * an M_IOCDATA message.
9055  */
9056 void
9057 ip_sioctl_copyin_setup(queue_t *q, mblk_t *mp)
9058 {
9059         int     copyin_size;
9060         struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
9061         ip_ioctl_cmd_t *ipip;
9062         cred_t *cr;
9063         ip_stack_t      *ipst;

9065         if (CONN_Q(q))
9066                 ipst = CONNQ_TO_IPST(q);
9067         else
9068                 ipst = ILLQ_TO_IPST(q);

9070         ipip = ip_sioctl_lookup(iocp->ioc_cmd);
9071         if (ipip == NULL) {
9072                 /*
9073                  * The ioctl is not one we understand or own.
9074                  * Pass it along to be processed down stream,
9075                  * if this is a module instance of IP, else nak
9076                  * the ioctl.
9077                  */
9078                 if (q->q_next == NULL) {
9079                         goto nak;
9080                 } else {
9081                         putnext(q, mp);
9082                         return;
9083                 }
9084         }

9086         /*
9087          * If this is deferred, then we will do all the checks when we
9088          * come back.
9089          */
9090         if ((iocp->ioc_cmd == SIOCGDSTINFO ||
9091             iocp->ioc_cmd == SIOCGIP6ADDRPOLICY) && !ip6_asp_can_lookup(ipst)) {
9092                 ip6_asp_pending_op(q, mp, ip_sioctl_copyin_resume);
9093                 return;
9094         }

9096         /*
9097          * Only allow a very small subset of IP ioctls on this stream if
9098          * IP is a module and not a driver. Allowing ioctls to be processed
9099          * in this case may cause assert failures or data corruption.
9100          * Typically G[L]IFFLAGS, SLIFNAME/IF_UNITSEL are the only few
9101          * ioctls allowed on an IP module stream, after which this stream
9102          * normally becomes a multiplexor (at which time the stream head
9103          * will fail all ioctls).
```

```
9104          */
9105         if ((q->q_next != NULL) && !(ipip->ipi_flags & IPI_MODOK)) {
9106                 goto nak;
9107         }
9108
9109         /* Make sure we have ioctl data to process. */
9110         if (mp->b_cont == NULL && !(ipip->ipi_flags & IPI_NULL_BCONT))
9111                 goto nak;
9112
9113         /*
9114          * Prefer dblk credential over ioctl credential; some synthesized
9115          * ioctls have kcred set because there's no way to crhold()
9116          * a credential in some contexts.  (ioc_cr is not crfree() by
9117          * the framework; the caller of ioctl needs to hold the reference
9118          * for the duration of the call).
9119          */
9120         cr = msg_getcred(mp, NULL);
9121         if (cr == NULL)
9122                 cr = iocp->ioc_cr;
9123
9124         /* Make sure normal users don't send down privileged ioctls */
9125         if ((ipip->ipi_flags & IPI_PRIV) &&
9126             (cr != NULL) && secpolicy_ip_config(cr, B_TRUE) != 0) {
9127                 /* We checked the privilege earlier but log it here */
9128                 miocnak(q, mp, 0, secpolicy_ip_config(cr, B_FALSE));
9129                 return;
9130         }
9131
9132         /*
9133          * The ioctl command tables can only encode fixed length
9134          * ioctl data. If the length is variable, the table will
9135          * encode the length as zero. Such special cases are handled
9136          * below in the switch.
9137          */
9138         if (ipip->ipi_copyin_size != 0) {
9139                 mi_copyin(q, mp, NULL, ipip->ipi_copyin_size);
9140                 return;
9141         }
9142
9143         switch (iocp->ioc_cmd) {
9144         case O_SIOCGIFCONF:
9145         case SIOCGIFCONF:
9146                 /*
9147                  * This IOCTL is hilarious.  See comments in
9148                  * ip_sioctl_get_ifconf for the story.
9149                  */
9150                 if (iocp->ioc_count == TRANSPARENT)
9151                         copyin_size = SIZEOF_STRUCT(ifconf,
9152                             iocp->ioc_flag);
9153                 else
9154                         copyin_size = iocp->ioc_count;
9155                 mi_copyin(q, mp, NULL, copyin_size);
9156                 return;
9157
9158         case O_SIOCGLIFCONF:
9159         case SIOCGLIFCONF:
9160                 copyin_size = SIZEOF_STRUCT(lifconf, iocp->ioc_flag);
9161                 mi_copyin(q, mp, NULL, copyin_size);
9162                 return;
9163
9164         case SIOCGLIFSRCOF:
9165                 copyin_size = SIZEOF_STRUCT(lifsrcof, iocp->ioc_flag);
9166                 mi_copyin(q, mp, NULL, copyin_size);
9167                 return;
9168
9169         case SIOCGIP6ADDRPOLICY:
```



```
9170                 ip_sioctl_ip6addrpolicy(q, mp);
9171                 ip6_asp_table_refrele(ipst);
9172                 return;
9173
9174         case SIOCSIP6ADDRPOLICY:
9175                 ip_sioctl_ip6addrpolicy(q, mp);
9176                 return;
9177
9178         case SIOCGDSTINFO:
9179                 ip_sioctl_dstinfo(q, mp);
9180                 ip6_asp_table_refrele(ipst);
9181                 return;
9182
9183         case ND_SET:
9184         case ND_GET:
9185                 ip_process_legacy_nddprop(q, mp);
9186                 return;
9187
9188         case SIOCSETPROP:
9189         case SIOCGETPROP:
9190                 ip_sioctl_getsetprop(q, mp);
9191                 return;
9192
9193         case I_PLINK:
9194         case I_PUNLINK:
9195         case I_LINK:
9196         case I_UNLINK:
9197                 /*
9198                  * We treat non-persistent link similarly as the persistent
9199                  * link case, in terms of plumbing/unplumbing, as well as
9200                  * dynamic re-plumbing events indicator.  See comments
9201                  * in ip_sioctl_plink() for more.
9202                  *
9203                  * Request can be enqueued in the 'ipsq' while waiting
9204                  * to become exclusive. So bump up the conn ref.
9205                  */
9206                 if (CONN_Q(q)) {
9207                         CONN_INC_REF(Q_TO_CONN(q));
9208                         CONN_INC_IOCTLREF(Q_TO_CONN(q))
9209                 }
9210                 ip_sioctl_plink(NULL, q, mp, NULL);
9211                 return;
9212
9213         case IP_IOCTL:
9214                 ip_wput_ioctl(q, mp);
9215                 return;
9216
9217         case SIOCILB:
9218                 /* The ioctl length varies depending on the ILB command. */
9219                 copyin_size = iocp->ioc_count;
9220                 if (copyin_size < sizeof (ilb_cmd_t))
9221                         goto nak;
9222                 mi_copyin(q, mp, NULL, copyin_size);
9223                 return;
9224
9225         default:
9226                 cmn_err(CE_PANIC, "should not happen ");
9227         }
9228 nak:
9229         if (mp->b_cont != NULL) {
9230                 freemsg(mp->b_cont);
9231                 mp->b_cont = NULL;
9232         }
9233         iocp->ioc_error = EINVAL;
9234         mp->b_datap->db_type = M_IOCNAK;
9235         iocp->ioc_count = 0;
```

```
9236         qreply(q, mp);
9237 }

9239 static void
9240 ip_sioctl_garp_reply(mblk_t *mp, ill_t *ill, void *hwaddr, int flags)
9241 {
9242         struct arpreq *ar;
9243         struct xarpreq *xar;
9244         mblk_t  *tmp;
9245         struct iocblk *iocp;
9246         int x_arp_ioctl = B_FALSE;
9247         int *flagsp;
9248         char *storage = NULL;

9250         ASSERT(ill != NULL);

9252         iocp = (struct iocblk *)mp->b_rptr;
9253         ASSERT(iocp->ioc_cmd == SIOCGXARP || iocp->ioc_cmd == SIOCGARP);

9255         tmp = (mp->b_cont)->b_cont; /* xarpreq/arpreq */
9256         if ((iocp->ioc_cmd == SIOCGXARP) ||
9257             (iocp->ioc_cmd == SIOCSXARP)) {
9258                 x_arp_ioctl = B_TRUE;
9259                 xar = (struct xarpreq *)tmp->b_rptr;
9260                 flagsp = &xar->xarp_flags;
9261                 storage = xar->xarp_ha.sdl_data;
9262         } else {
9263                 ar = (struct arpreq *)tmp->b_rptr;
9264                 flagsp = &ar->arp_flags;
9265                 storage = ar->arp_ha.sa_data;
9266         }

9268         /*
9269          * We're done if this is not an SIOCG{X}ARP
9270          */
9271         if (x_arp_ioctl) {
9272                 storage += ill_xarp_info(&xar->xarp_ha, ill);
9273                 if ((ill->ill_phys_addr_length + ill->ill_name_length) >
9274                     sizeof (xar->xarp_ha.sdl_data)) {
9275                         iocp->ioc_error = EINVAL;
9276                         return;
9277                 }
9278         }
9279         *flagsp = ATF_INUSE;
9280         /*
9281          * If /sbin/arp told us we are the authority using the "permanent"
9282          * flag, or if this is one of my addresses print "permanent"
9283          * in the /sbin/arp output.
9284          */
9285         if ((flags & NCE_F_MYADDR) || (flags & NCE_F_AUTHORITY))
9286                 *flagsp |= ATF_AUTHORITY;
9287         if (flags & NCE_F_NONUD)
9288                 *flagsp |= ATF_PERM; /* not subject to aging */
9289         if (flags & NCE_F_PUBLISH)
9290                 *flagsp |= ATF_PUBL;
9291         if (hwaddr != NULL) {
9292                 *flagsp |= ATF_COM;
9293                 bcopy((char *)hwaddr, storage, ill->ill_phys_addr_length);
9294         }
9295 }

9297 /*
9298  * Create a new logical interface. If ipif_id is zero (i.e. not a logical
9299  * interface) create the next available logical interface for this
9300  * physical interface.
9301  * If ipif is NULL (i.e. the lookup didn't find one) attempt to create an
```



```
9302  * ipif with the specified name.
9303  *
9304  * If the address family is not AF_UNSPEC then set the address as well.
9305  *
9306  * If ip_sioctl_addr returns EINPROGRESS then the ioctl (the copyout)
9307  * is completed when the DL_BIND_ACK arrive in ip_rput_dlpi_writer.
9308  *
9309  * Executed as a writer on the ill.
9310  * So no lock is needed to traverse the ipif chain, or examine the
9311  * phyint flags.
9312  */
9313 /* ARGSUSED */
9314 int
9315 ip_sioctl_addif(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
9316     ip_ioctl_cmd_t *dummy_ipip, void *dummy_ifreq)
9317 {
9318         mblk_t  *mp1;
9319         struct lifreq *lifr;
9320         boolean_t       isv6;
9321         boolean_t       exists;
9322         char    *name;
9323         char    *endp;
9324         char    *cp;
9325         int     namelen;
9326         ipif_t  *ipif;
9327         long    id;
9328         ipsq_t  *ipsq;
9329         ill_t   *ill;
9330         sin_t   *sin;
9331         int     err = 0;
9332         boolean_t found_sep = B_FALSE;
9333         conn_t  *connp;
9334         zoneid_t zoneid;
9335         ip_stack_t *ipst = CONNQ_TO_IPST(q);

9337         ASSERT(q->q_next == NULL);
9338         ip1dbg(("ip_sioctl_addif\n"));
9339         /* Existence of mp1 has been checked in ip_wput_nondata */
9340         mp1 = mp->b_cont->b_cont;
9341         /*
9342          * Null terminate the string to protect against buffer
9343          * overrun. String was generated by user code and may not
9344          * be trusted.
9345          */
9346         lifr = (struct lifreq *)mp1->b_rptr;
9347         lifr->lifr_name[LIFNAMSIZ - 1] = '\0';
9348         name = lifr->lifr_name;
9349         ASSERT(CONN_Q(q));
9350         connp = Q_TO_CONN(q);
9351         isv6 = (connp->conn_family == AF_INET6);
9352         zoneid = connp->conn_zoneid;
9353         namelen = mi_strlen(name);
9354         if (namelen == 0)
9355                 return (EINVAL);

9357         exists = B_FALSE;
9358         if ((namelen + 1 == sizeof (ipif_loopback_name)) &&
9359             (mi_strcmp(name, ipif_loopback_name) == 0)) {
9360                 /*
9361                  * Allow creating lo0 using SIOCLIFADDIF.
9362                  * can't be any other writer thread. So can pass null below
9363                  * for the last 4 args to ipif_lookup_name.
9364                  */
9365                 ipif = ipif_lookup_on_name(lifr->lifr_name, namelen, B_TRUE,
9366                     &exists, isv6, zoneid, ipst);
9367                 /* Prevent any further action */
```

```
9368                if (ipif == NULL) {
9369                        return (ENOBUFS);
9370                } else if (!exists) {
9371                        /* We created the ipif now and as writer */
9372                        ipif_refrele(ipif);
9373                        return (0);
9374                } else {
9375                        ill = ipif->ipif_ill;
9376                        ill_refhold(ill);
9377                        ipif_refrele(ipif);
9378                }
9379        } else {
9380                /* Look for a colon in the name. */
9381                endp = &name[namelen];
9382                for (cp = endp; --cp > name; ) {
9383                        if (*cp == IPIF_SEPARATOR_CHAR) {
9384                                found_sep = B_TRUE;
9385                                /*
9386                                 * Reject any non-decimal aliases for plumbing
9387                                 * of logical interfaces. Aliases with leading
9388                                 * zeroes are also rejected as they introduce
9389                                 * ambiguity in the naming of the interfaces.
9390                                 * Comparing with "0" takes care of all such
9391                                 * cases.
9392                                 */
9393                                if ((strncmp("0", cp+1, 1)) == 0)
9394                                        return (EINVAL);

9396                                if (ddi_strtol(cp+1, &endp, 10, &id) != 0 ||
9397                                    id <= 0 || *endp != '\0') {
9398                                        return (EINVAL);
9399                                }
9400                                *cp = '\0';
9401                                break;
9402                        }
9403                }
9404                ill = ill_lookup_on_name(name, B_FALSE, isv6, NULL, ipst);
9405                if (found_sep)
9406                        *cp = IPIF_SEPARATOR_CHAR;
9407                if (ill == NULL)
9408                        return (ENXIO);
9409        }

9411        ipsq = ipsq_try_enter(NULL, ill, q, mp, ip_process_ioctl, NEW_OP,
9412            B_TRUE);

9414        /*
9415         * Release the refhold due to the lookup, now that we are excl
9416         * or we are just returning
9417         */
9418        ill_refrele(ill);

9420        if (ipsq == NULL)
9421                return (EINPROGRESS);

9423        /* We are now exclusive on the IPSQ */
9424        ASSERT(IAM_WRITER_ILL(ill));

9426        if (found_sep) {
9427                /* Now see if there is an IPIF with this unit number. */
9428                for (ipif = ill->ill_ipif; ipif != NULL;
9429                    ipif = ipif->ipif_next) {
9430                        if (ipif->ipif_id == id) {
9431                                err = EEXIST;
9432                                goto done;
9433                        }
```



```
9434                }
9435        }

9437        /*
9438         * We use IRE_LOCAL for lo0:1 etc. for "receive only" use
9439         * of lo0.  Plumbing for lo0:0 happens in ipif_lookup_on_name()
9440         * instead.
9441         */
9442        if ((ipif = ipif_allocate(ill, found_sep ? id : -1, IRE_LOCAL,
9443            B_TRUE, B_TRUE, &err)) == NULL) {
9444                goto done;
9445        }

9447        /* Return created name with ioctl */
9448        (void) sprintf(lifr->lifr_name, "%s%c%d", ill->ill_name,
9449            IPIF_SEPARATOR_CHAR, ipif->ipif_id);
9450        ip1dbg(("created %s\n", lifr->lifr_name));

9452        /* Set address */
9453        sin = (sin_t *)&lifr->lifr_addr;
9454        if (sin->sin_family != AF_UNSPEC) {
9455                err = ip_sioctl_addr(ipif, sin, q, mp,
9456                    &ip_ndx_ioctl_table[SIOCLIFADDR_NDX], lifr);
9457        }

9459 done:
9460        ipsq_exit(ipsq);
9461        return (err);
9462 }

9464 /*
9465  * Remove an existing logical interface. If ipif_id is zero (i.e. not a logical
9466  * interface) delete it based on the IP address (on this physical interface).
9467  * Otherwise delete it based on the ipif_id.
9468  * Also, special handling to allow a removeif of lo0.
9469  */
9470 /* ARGSUSED */
9471 int
9472 ip_sioctl_removeif(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9473     ip_ioctl_cmd_t *ipip, void *dummy_if_req)
9474 {
9475        conn_t          *connp;
9476        ill_t           *ill = ipif->ipif_ill;
9477        boolean_t        success;
9478        ip_stack_t      *ipst;

9480        ipst = CONNQ_TO_IPST(q);

9482        ASSERT(q->q_next == NULL);
9483        ip1dbg(("ip_sioctl_remove_if(%s:%u %p)\n",
9484            ill->ill_name, ipif->ipif_id, (void *)ipif));
9485        ASSERT(IAM_WRITER_IPIF(ipif));

9487        connp = Q_TO_CONN(q);
9488        /*
9489         * Special case for unplumbing lo0 (the loopback physical interface).
9490         * If unplumbing lo0, the incoming address structure has been
9491         * initialized to all zeros. When unplumbing lo0, all its logical
9492         * interfaces must be removed too.
9493         *
9494         * Note that this interface may be called to remove a specific
9495         * loopback logical interface (eg, lo0:1). But in that case
9496         * ipif->ipif_id != 0 so that the code path for that case is the
9497         * same as any other interface (meaning it skips the code directly
9498         * below).
9499         */
```

```
9500         if (ipif->ipif_id == 0 && ill->ill_net_type == IRE_LOOPBACK) {
9501                 if (sin->sin_family == AF_UNSPEC &&
9502                     (IN6_IS_ADDR_UNSPECIFIED(&((sin6_t *)sin)->sin6_addr))) {
9503                         /*
9504                          * Mark it condemned. No new ref. will be made to ill.
9505                          */
9506                         mutex_enter(&ill->ill_lock);
9507                         ill->ill_state_flags |= ILL_CONDEMNED;
9508                         for (ipif = ill->ill_ipif; ipif != NULL;
9509                             ipif = ipif->ipif_next) {
9510                                 ipif->ipif_state_flags |= IPIF_CONDEMNED;
9511                         }
9512                         mutex_exit(&ill->ill_lock);

9514                         ipif = ill->ill_ipif;
9515                         /* unplumb the loopback interface */
9516                         ill_delete(ill);
9517                         mutex_enter(&connp->conn_lock);
9518                         mutex_enter(&ill->ill_lock);

9520                         /* Are any references to this ill active */
9521                         if (ill_is_freeable(ill)) {
9522                                 mutex_exit(&ill->ill_lock);
9523                                 mutex_exit(&connp->conn_lock);
9524                                 ill_delete_tail(ill);
9525                                 mi_free(ill);
9526                                 return (0);
9527                         }
9528                         success = ipsq_pending_mp_add(connp, ipif,
9529                             CONNP_TO_WQ(connp), mp, ILL_FREE);
9530                         mutex_exit(&connp->conn_lock);
9531                         mutex_exit(&ill->ill_lock);
9532                         if (success)
9533                                 return (EINPROGRESS);
9534                         else
9535                                 return (EINTR);
9536                 }
9537         }

9539         if (ipif->ipif_id == 0) {
9540                 ipsq_t *ipsq;

9542                 /* Find based on address */
9543                 if (ipif->ipif_isv6) {
9544                         sin6_t *sin6;

9546                         if (sin->sin_family != AF_INET6)
9547                                 return (EAFNOSUPPORT);

9549                         sin6 = (sin6_t *)sin;
9550                         /* We are a writer, so we should be able to lookup */
9551                         ipif = ipif_lookup_addr_exact_v6(&sin6->sin6_addr, ill,
9552                             ipst);
9553                 } else {
9554                         if (sin->sin_family != AF_INET)
9555                                 return (EAFNOSUPPORT);

9557                         /* We are a writer, so we should be able to lookup */
9558                         ipif = ipif_lookup_addr_exact(sin->sin_addr.s_addr, ill,
9559                             ipst);
9560                 }
9561                 if (ipif == NULL) {
9562                         return (EADDRNOTAVAIL);
9563                 }

9565                 /*
```



```
9566                  * It is possible for a user to send an SIOCLIFREMOVEIF with
9567                  * lifr_name of the physical interface but with an ip address
9568                  * lifr_addr of a logical interface plumbed over it.
9569                  * So update ipx_current_ipif now that ipif points to the
9570                  * correct one.
9571                  */
9572                 ipsq = ipif->ipif_ill->ill_phyint->phyint_ipsq;
9573                 ipsq->ipsq_xop->ipx_current_ipif = ipif;

9575                 /* This is a writer */
9576                 ipif_refrele(ipif);
9577         }

9579         /*
9580          * Can not delete instance zero since it is tied to the ill.
9581          */
9582         if (ipif->ipif_id == 0)
9583                 return (EBUSY);

9585         mutex_enter(&ill->ill_lock);
9586         ipif->ipif_state_flags |= IPIF_CONDEMNED;
9587         mutex_exit(&ill->ill_lock);

9589         ipif_free(ipif);

9591         mutex_enter(&connp->conn_lock);
9592         mutex_enter(&ill->ill_lock);

9594         /* Are any references to this ipif active */
9595         if (ipif_is_freeable(ipif)) {
9596                 mutex_exit(&ill->ill_lock);
9597                 mutex_exit(&connp->conn_lock);
9598                 ipif_non_duplicate(ipif);
9599                 (void) ipif_down_tail(ipif);
9600                 ipif_free_tail(ipif); /* frees ipif */
9601                 return (0);
9602         }
9603         success = ipsq_pending_mp_add(connp, ipif, CONNP_TO_WQ(connp), mp,
9604             IPIF_FREE);
9605         mutex_exit(&ill->ill_lock);
9606         mutex_exit(&connp->conn_lock);
9607         if (success)
9608                 return (EINPROGRESS);
9609         else
9610                 return (EINTR);
9611 }

9613 /*
9614  * Restart the removeif ioctl. The refcnt has gone down to 0.
9615  * The ipif is already condemned. So can't find it thru lookups.
9616  */
9617 /* ARGSUSED */
9618 int
9619 ip_sioctl_removeif_restart(ipif_t *ipif, sin_t *dummy_sin, queue_t *q,
9620     mblk_t *mp, ip_ioctl_cmd_t *ipip, void *dummy_if_req)
9621 {
9622         ill_t *ill = ipif->ipif_ill;

9624         ASSERT(IAM_WRITER_IPIF(ipif));
9625         ASSERT(ipif->ipif_state_flags & IPIF_CONDEMNED);

9627         ip1dbg(("ip_sioctl_removeif_restart(%s:%u %p)\n",
9628             ill->ill_name, ipif->ipif_id, (void *)ipif));

9630         if (ipif->ipif_id == 0 && ill->ill_net_type == IRE_LOOPBACK) {
9631                 ASSERT(ill->ill_state_flags & ILL_CONDEMNED);
```

```
9632                ill_delete_tail(ill);
9633                mi_free(ill);
9634                return (0);
9635        }

9637        ipif_non_duplicate(ipif);
9638        (void) ipif_down_tail(ipif);
9639        ipif_free_tail(ipif);

9641        return (0);
9642 }

9644 /*
9645  * Set the local interface address using the given prefix and ill_token.
9646  */
9647 /* ARGSUSED */
9648 int
9649 ip_sioctl_prefix(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9650     ip_ioctl_cmd_t *dummy_ipip, void *dummy_ifreq)
9651 {
9652        int err;
9653        in6_addr_t v6addr;
9654        sin6_t *sin6;
9655        ill_t *ill;
9656        int i;

9658        ip1dbg(("ip_sioctl_prefix(%s:%u %p)\n",
9659            ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

9661        ASSERT(IAM_WRITER_IPIF(ipif));

9663        if (!ipif->ipif_isv6)
9664                return (EINVAL);

9666        if (sin->sin_family != AF_INET6)
9667                return (EAFNOSUPPORT);

9669        sin6 = (sin6_t *)sin;
9670        v6addr = sin6->sin6_addr;
9671        ill = ipif->ipif_ill;

9673        if (IN6_IS_ADDR_UNSPECIFIED(&v6addr) ||
9674            IN6_IS_ADDR_UNSPECIFIED(&ill->ill_token))
9675                return (EADDRNOTAVAIL);

9677        for (i = 0; i < 4; i++)
9678                sin6->sin6_addr.s6_addr32[i] |= ill->ill_token.s6_addr32[i];

9680        err = ip_sioctl_addr(ipif, sin, q, mp,
9681            &ip_ndx_ioctl_table[SIOCLIFADDR_NDX], dummy_ifreq);
9682        return (err);
9683 }

9685 /*
9686  * Restart entry point to restart the address set operation after the
9687  * refcounts have dropped to zero.
9688  */
9689 /* ARGSUSED */
9690 int
9691 ip_sioctl_prefix_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9692     ip_ioctl_cmd_t *ipip, void *ifreq)
9693 {
9694        ip1dbg(("ip_sioctl_prefix_restart(%s:%u %p)\n",
9695            ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
9696        return (ip_sioctl_addr_restart(ipif, sin, q, mp, ipip, ifreq));
9697 }
```



```
9699 /*
9700  * Set the local interface address.
9701  * Allow an address of all zero when the interface is down.
9702  */
9703 /* ARGSUSED */
9704 int
9705 ip_sioctl_addr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9706     ip_ioctl_cmd_t *dummy_ipip, void *dummy_ifreq)
9707 {
9708        int err = 0;
9709        in6_addr_t v6addr;
9710        boolean_t need_up = B_FALSE;
9711        ill_t *ill;
9712        int i;

9714        ip1dbg(("ip_sioctl_addr(%s:%u %p)\n",
9715            ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

9717        ASSERT(IAM_WRITER_IPIF(ipif));

9719        ill = ipif->ipif_ill;
9720        if (ipif->ipif_isv6) {
9721                sin6_t *sin6;
9722                phyint_t *phyi;

9724                if (sin->sin_family != AF_INET6)
9725                        return (EAFNOSUPPORT);

9727                sin6 = (sin6_t *)sin;
9728                v6addr = sin6->sin6_addr;
9729                phyi = ill->ill_phyint;

9731                /*
9732                 * Enforce that true multicast interfaces have a link-local
9733                 * address for logical unit 0.
9734                 *
9735                 * However for those ipif's for which link-local address was
9736                 * not created by default, also allow setting :: as the address.
9737                 * This scenario would arise, when we delete an address on ipif
9738                 * with logical unit 0, we would want to set :: as the address.
9739                 */
9740                if (ipif->ipif_id == 0 &&
9741                    (ill->ill_flags & ILLF_MULTICAST) &&
9742                    !(ipif->ipif_flags & (IPIF_POINTOPOINT)) &&
9743                    !(phyi->phyint_flags & (PHYI_LOOPBACK)) &&
9744                    !IN6_IS_ADDR_LINKLOCAL(&v6addr)) {

9746                        /*
9747                         * if default link-local was not created by kernel for
9748                         * this ill, allow setting :: as the address on ipif:0.
9749                         */
9750                        if (ill->ill_flags & ILLF_NOLINKLOCAL) {
9751                                if (!IN6_IS_ADDR_UNSPECIFIED(&v6addr))
9752                                        return (EADDRNOTAVAIL);
9753                        } else {
9754                                return (EADDRNOTAVAIL);
9755                        }
9756                }

9758                /*
9759                 * up interfaces shouldn't have the unspecified address
9760                 * unless they also have the IPIF_NOLOCAL flags set and
9761                 * have a subnet assigned.
9762                 */
9763                if ((ipif->ipif_flags & IPIF_UP) &&
```

```
                    IN6_IS_ADDR_UNSPECIFIED(&v6addr) &&
                    (!(ipif->ipif_flags & IPIF_NOLOCAL) ||
                    IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6subnet))) {
                        return (EADDRNOTAVAIL);
                }

                if (!ip_local_addr_ok_v6(&v6addr, &ipif->ipif_v6net_mask))
                        return (EADDRNOTAVAIL);
        } else {
                ipaddr_t addr;

                if (sin->sin_family != AF_INET)
                        return (EAFNOSUPPORT);

                addr = sin->sin_addr.s_addr;

                /* Allow INADDR_ANY as the local address. */
                if (addr != INADDR_ANY &&
                    !ip_addr_ok_v4(addr, ipif->ipif_net_mask))
                        return (EADDRNOTAVAIL);

                IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
        }
        /*
         * verify that the address being configured is permitted by the
         * ill_allowed_ips[] for the interface.
         */
        if (ill->ill_allowed_ips_cnt > 0) {
                for (i = 0; i < ill->ill_allowed_ips_cnt; i++) {
                        if (IN6_ARE_ADDR_EQUAL(&ill->ill_allowed_ips[i],
                            &v6addr))
                                break;
                }
                if (i == ill->ill_allowed_ips_cnt) {
                        pr_addr_dbg("!allowed addr %s\n", AF_INET6, &v6addr);
                        return (EPERM);
                }
        }
        /*
         * Even if there is no change we redo things just to rerun
         * ipif_set_default.
         */
        if (ipif->ipif_flags & IPIF_UP) {
                /*
                 * Setting a new local address, make sure
                 * we have net and subnet bcast ire's for
                 * the old address if we need them.
                 */
                /*
                 * If the interface is already marked up,
                 * we call ipif_down which will take care
                 * of ditching any IREs that have been set
                 * up based on the old interface address.
                 */
                err = ipif_logical_down(ipif, q, mp);
                if (err == EINPROGRESS)
                        return (err);
                (void) ipif_down_tail(ipif);
                need_up = 1;
        }

        err = ip_sioctl_addr_tail(ipif, sin, q, mp, need_up);
        return (err);
}

int
```



```
ip_sioctl_addr_tail(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
    boolean_t need_up)
{
        in6_addr_t v6addr;
        in6_addr_t ov6addr;
        ipaddr_t addr;
        sin6_t  *sin6;
        int     sinlen;
        int     err = 0;
        ill_t   *ill = ipif->ipif_ill;
        boolean_t need_dl_down;
        boolean_t need_arp_down;
        struct iocblk *iocp;

        iocp = (mp != NULL) ? (struct iocblk *)mp->b_rptr : NULL;

        ip1dbg(("ip_sioctl_addr_tail(%s:%u %p)\n",
            ill->ill_name, ipif->ipif_id, (void *)ipif));
        ASSERT(IAM_WRITER_IPIF(ipif));

        /* Must cancel any pending timer before taking the ill_lock */
        if (ipif->ipif_recovery_id != 0)
                (void) untimeout(ipif->ipif_recovery_id);
        ipif->ipif_recovery_id = 0;

        if (ipif->ipif_isv6) {
                sin6 = (sin6_t *)sin;
                v6addr = sin6->sin6_addr;
                sinlen = sizeof (struct sockaddr_in6);
        } else {
                addr = sin->sin_addr.s_addr;
                IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
                sinlen = sizeof (struct sockaddr_in);
        }
        mutex_enter(&ill->ill_lock);
        ov6addr = ipif->ipif_v6lcl_addr;
        ipif->ipif_v6lcl_addr = v6addr;
        sctp_update_ipif_addr(ipif, ov6addr);
        ipif->ipif_addr_ready = 0;

        ip_rts_newaddrmsg(RTM_CHGADDR, 0, ipif, RTSQ_DEFAULT);

        /*
         * If the interface was previously marked as a duplicate, then since
         * we've now got a "new" address, it should no longer be considered a
         * duplicate -- even if the "new" address is the same as the old one.
         * Note that if all ipifs are down, we may have a pending ARP down
         * event to handle.  This is because we want to recover from duplicates
         * and thus delay tearing down ARP until the duplicates have been
         * removed or disabled.
         */
        need_dl_down = need_arp_down = B_FALSE;
        if (ipif->ipif_flags & IPIF_DUPLICATE) {
                need_arp_down = !need_up;
                ipif->ipif_flags &= ~IPIF_DUPLICATE;
                if (--ill->ill_ipif_dup_count == 0 && !need_up &&
                    ill->ill_ipif_up_count == 0 && ill->ill_dl_up) {
                        need_dl_down = B_TRUE;
                }
        }

        ipif_set_default(ipif);

        /*
         * If we've just manually set the IPv6 link-local address (0th ipif),
         * tag the ill so that future updates to the interface ID don't result
```

```
9896          * in this address getting automatically reconfigured from under the
9897          * administrator.
9898          */
9899         if (ipif->ipif_isv6 && ipif->ipif_id == 0) {
9900                 if (iocp == NULL || (iocp->ioc_cmd == SIOCSLIFADDR &&
9901                     !IN6_IS_ADDR_UNSPECIFIED(&v6addr)))
9902                         ill->ill_manual_linklocal = 1;
9903         }

9905         /*
9906          * When publishing an interface address change event, we only notify
9907          * the event listeners of the new address.  It is assumed that if they
9908          * actively care about the addresses assigned that they will have
9909          * already discovered the previous address assigned (if there was one.)
9910          *
9911          * Don't attach nic event message for SIOCLIFADDIF ioctl.
9912          */
9913         if (iocp != NULL && iocp->ioc_cmd != SIOCLIFADDIF) {
9914                 ill_nic_event_dispatch(ill, MAP_IPIF_ID(ipif->ipif_id),
9915                     NE_ADDRESS_CHANGE, sin, sinlen);
9916         }

9918         mutex_exit(&ill->ill_lock);

9920         if (need_up) {
9921                 /*
9922                  * Now bring the interface back up.  If this
9923                  * is the only IPIF for the ILL, ipif_up
9924                  * will have to re-bind to the device, so
9925                  * we may get back EINPROGRESS, in which
9926                  * case, this IOCTL will get completed in
9927                  * ip_rput_dlpi when we see the DL_BIND_ACK.
9928                  */
9929                 err = ipif_up(ipif, q, mp);
9930         } else {
9931                 /* Perhaps ilgs should use this ill */
9932                 update_conn_ill(NULL, ill->ill_ipst);
9933         }

9935         if (need_dl_down)
9936                 ill_dl_down(ill);

9938         if (need_arp_down && !ill->ill_isv6)
9939                 (void) ipif_arp_down(ipif);

9941         /*
9942          * The default multicast interface might have changed (for
9943          * instance if the IPv6 scope of the address changed)
9944          */
9945         ire_increment_multicast_generation(ill->ill_ipst, ill->ill_isv6);

9947         return (err);
9948 }

9950 /*
9951  * Restart entry point to restart the address set operation after the
9952  * refcounts have dropped to zero.
9953  */
9954 /* ARGSUSED */
9955 int
9956 ip_sioctl_addr_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9957     ip_ioctl_cmd_t *ipip, void *ifreq)
9958 {
9959         ip1dbg(("ip_sioctl_addr_restart(%s:%u %p)\n",
9960             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
9961         ASSERT(IAM_WRITER_IPIF(ipif));
```



```
9962         (void) ipif_down_tail(ipif);
9963         return (ip_sioctl_addr_tail(ipif, sin, q, mp, B_TRUE));
9964 }

9966 /* ARGSUSED */
9967 int
9968 ip_sioctl_get_addr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9969     ip_ioctl_cmd_t *ipip, void *if_req)
9970 {
9971         sin6_t *sin6 = (struct sockaddr_in6 *)sin;
9972         struct lifreq *lifr = (struct lifreq *)if_req;

9974         ip1dbg(("ip_sioctl_get_addr(%s:%u %p)\n",
9975             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
9976         /*
9977          * The net mask and address can't change since we have a
9978          * reference to the ipif. So no lock is necessary.
9979          */
9980         if (ipif->ipif_isv6) {
9981                 *sin6 = sin6_null;
9982                 sin6->sin6_family = AF_INET6;
9983                 sin6->sin6_addr = ipif->ipif_v6lcl_addr;
9984                 ASSERT(ipip->ipi_cmd_type == LIF_CMD);
9985                 lifr->lifr_addrlen =
9986                     ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
9987         } else {
9988                 *sin = sin_null;
9989                 sin->sin_family = AF_INET;
9990                 sin->sin_addr.s_addr = ipif->ipif_lcl_addr;
9991                 if (ipip->ipi_cmd_type == LIF_CMD) {
9992                         lifr->lifr_addrlen =
9993                             ip_mask_to_plen(ipif->ipif_net_mask);
9994                 }
9995         }
9996         return (0);
9997 }

9999 /*
10000  * Set the destination address for a pt-pt interface.
10001  */
10002 /* ARGSUSED */
10003 int
10004 ip_sioctl_dstaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10005     ip_ioctl_cmd_t *ipip, void *if_req)
10006 {
10007         int err = 0;
10008         in6_addr_t v6addr;
10009         boolean_t need_up = B_FALSE;

10011         ip1dbg(("ip_sioctl_dstaddr(%s:%u %p)\n",
10012             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10013         ASSERT(IAM_WRITER_IPIF(ipif));

10015         if (ipif->ipif_isv6) {
10016                 sin6_t *sin6;

10018                 if (sin->sin_family != AF_INET6)
10019                         return (EAFNOSUPPORT);

10021                 sin6 = (sin6_t *)sin;
10022                 v6addr = sin6->sin6_addr;

10024                 if (!ip_remote_addr_ok_v6(&v6addr, &ipif->ipif_v6net_mask))
10025                         return (EADDRNOTAVAIL);
10026         } else {
10027                 ipaddr_t addr;
```

```
10029                 if (sin->sin_family != AF_INET)
10030                         return (EAFNOSUPPORT);

10032                 addr = sin->sin_addr.s_addr;
10033                 if (addr != INADDR_ANY &&
10034                     !ip_addr_ok_v4(addr, ipif->ipif_net_mask)) {
10035                         return (EADDRNOTAVAIL);
10036                 }

10038                 IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
10039         }

10041         if (IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6pp_dst_addr, &v6addr))
10042                 return (0);     /* No change */

10044         if (ipif->ipif_flags & IPIF_UP) {
10045                 /*
10046                  * If the interface is already marked up,
10047                  * we call ipif_down which will take care
10048                  * of ditching any IREs that have been set
10049                  * up based on the old pp dst address.
10050                  */
10051                 err = ipif_logical_down(ipif, q, mp);
10052                 if (err == EINPROGRESS)
10053                         return (err);
10054                 (void) ipif_down_tail(ipif);
10055                 need_up = B_TRUE;
10056         }
10057         /*
10058          * could return EINPROGRESS. If so ioctl will complete in
10059          * ip_rput_dlpi_writer
10060          */
10061         err = ip_sioctl_dstaddr_tail(ipif, sin, q, mp, need_up);
10062         return (err);
10063 }

10065 static int
10066 ip_sioctl_dstaddr_tail(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10067     boolean_t need_up)
10068 {
10069         in6_addr_t v6addr;
10070         ill_t   *ill = ipif->ipif_ill;
10071         int     err = 0;
10072         boolean_t need_dl_down;
10073         boolean_t need_arp_down;

10075         ip1dbg(("ip_sioctl_dstaddr_tail(%s:%u %p)\n", ill->ill_name,
10076             ipif->ipif_id, (void *)ipif));

10078         /* Must cancel any pending timer before taking the ill_lock */
10079         if (ipif->ipif_recovery_id != 0)
10080                 (void) untimeout(ipif->ipif_recovery_id);
10081         ipif->ipif_recovery_id = 0;

10083         if (ipif->ipif_isv6) {
10084                 sin6_t *sin6;

10086                 sin6 = (sin6_t *)sin;
10087                 v6addr = sin6->sin6_addr;
10088         } else {
10089                 ipaddr_t addr;

10091                 addr = sin->sin_addr.s_addr;
10092                 IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
10093         }
```



```
10094         mutex_enter(&ill->ill_lock);
10095         /* Set point to point destination address. */
10096         if ((ipif->ipif_flags & IPIF_POINTOPOINT) == 0) {
10097                 /*
10098                  * Allow this as a means of creating logical
10099                  * pt-pt interfaces on top of e.g. an Ethernet.
10100                  * XXX Undocumented HACK for testing.
10101                  * pt-pt interfaces are created with NUD disabled.
10102                  */
10103                 ipif->ipif_flags |= IPIF_POINTOPOINT;
10104                 ipif->ipif_flags &= ~IPIF_BROADCAST;
10105                 if (ipif->ipif_isv6)
10106                         ill->ill_flags |= ILLF_NONUD;
10107         }

10109         /*
10110          * If the interface was previously marked as a duplicate, then since
10111          * we've now got a "new" address, it should no longer be considered a
10112          * duplicate -- even if the "new" address is the same as the old one.
10113          * Note that if all ipifs are down, we may have a pending ARP down
10114          * event to handle.
10115          */
10116         need_dl_down = need_arp_down = B_FALSE;
10117         if (ipif->ipif_flags & IPIF_DUPLICATE) {
10118                 need_arp_down = !need_up;
10119                 ipif->ipif_flags &= ~IPIF_DUPLICATE;
10120                 if (--ill->ill_ipif_dup_count == 0 && !need_up &&
10121                     ill->ill_ipif_up_count == 0 && ill->ill_dl_up) {
10122                         need_dl_down = B_TRUE;
10123                 }
10124         }

10126         /*
10127          * If we've just manually set the IPv6 destination link-local address
10128          * (0th ipif), tag the ill so that future updates to the destination
10129          * interface ID (as can happen with interfaces over IP tunnels) don't
10130          * result in this address getting automatically reconfigured from
10131          * under the administrator.
10132          */
10133         if (ipif->ipif_isv6 && ipif->ipif_id == 0)
10134                 ill->ill_manual_dst_linklocal = 1;

10136         /* Set the new address. */
10137         ipif->ipif_v6pp_dst_addr = v6addr;
10138         /* Make sure subnet tracks pp_dst */
10139         ipif->ipif_v6subnet = ipif->ipif_v6pp_dst_addr;
10140         mutex_exit(&ill->ill_lock);

10142         if (need_up) {
10143                 /*
10144                  * Now bring the interface back up.  If this
10145                  * is the only IPIF for the ILL, ipif_up
10146                  * will have to re-bind to the device, so
10147                  * we may get back EINPROGRESS, in which
10148                  * case, this IOCTL will get completed in
10149                  * ip_rput_dlpi when we see the DL_BIND_ACK.
10150                  */
10151                 err = ipif_up(ipif, q, mp);
10152         }

10154         if (need_dl_down)
10155                 ill_dl_down(ill);
10156         if (need_arp_down && !ipif->ipif_isv6)
10157                 (void) ipif_arp_down(ipif);

10159         return (err);
```

```
10160 }

10162 /*
10163  * Restart entry point to restart the dstaddress set operation after the
10164  * refcounts have dropped to zero.
10165  */
10166 /* ARGSUSED */
10167 int
10168 ip_sioctl_dstaddr_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10169     ip_ioctl_cmd_t *ipip, void *ifreq)
10170 {
10171         ip1dbg(("ip_sioctl_dstaddr_restart(%s:%u %p)\n",
10172             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10173         (void) ipif_down_tail(ipif);
10174         return (ip_sioctl_dstaddr_tail(ipif, sin, q, mp, B_TRUE));
10175 }

10177 /* ARGSUSED */
10178 int
10179 ip_sioctl_get_dstaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10180     ip_ioctl_cmd_t *ipip, void *if_req)
10181 {
10182         sin6_t  *sin6 = (struct sockaddr_in6 *)sin;

10184         ip1dbg(("ip_sioctl_get_dstaddr(%s:%u %p)\n",
10185             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10186         /*
10187          * Get point to point destination address. The addresses can't
10188          * change since we hold a reference to the ipif.
10189          */
10190         if ((ipif->ipif_flags & IPIF_POINTOPOINT) == 0)
10191                 return (EADDRNOTAVAIL);

10193         if (ipif->ipif_isv6) {
10194                 ASSERT(ipip->ipi_cmd_type == LIF_CMD);
10195                 *sin6 = sin6_null;
10196                 sin6->sin6_family = AF_INET6;
10197                 sin6->sin6_addr = ipif->ipif_v6pp_dst_addr;
10198         } else {
10199                 *sin = sin_null;
10200                 sin->sin_family = AF_INET;
10201                 sin->sin_addr.s_addr = ipif->ipif_pp_dst_addr;
10202         }
10203         return (0);
10204 }

10206 /*
10207  * Check which flags will change by the given flags being set
10208  * silently ignore flags which userland is not allowed to control.
10209  * (Because these flags may change between SIOCGLIFFLAGS and
10210  * SIOCSLIFFLAGS, and that's outside of userland's control,
10211  * we need to silently ignore them rather than fail.)
10212  */
10213 static void
10214 ip_sioctl_flags_onoff(ipif_t *ipif, uint64_t flags, uint64_t *onp,
10215     uint64_t *offp)
10216 {
10217         ill_t           *ill = ipif->ipif_ill;
10218         phyint_t        *phyi = ill->ill_phyint;
10219         uint64_t        cantchange_flags, intf_flags;
10220         uint64_t        turn_on, turn_off;

10222         intf_flags = ipif->ipif_flags | ill->ill_flags | phyi->phyint_flags;
10223         cantchange_flags = IFF_CANTCHANGE;
10224         if (IS_IPMP(ill))
10225                 cantchange_flags |= IFF_IPMP_CANTCHANGE;
```



```
10226         turn_on = (flags ^ intf_flags) & ~cantchange_flags;
10227         turn_off = intf_flags & turn_on;
10228         turn_on ^= turn_off;
10229         *onp = turn_on;
10230         *offp = turn_off;
10231 }

10233 /*
10234  * Set interface flags.  Many flags require special handling (e.g.,
10235  * bringing the interface down); see below for details.
10236  *
10237  * NOTE : We really don't enforce that ipif_id zero should be used
10238  *        for setting any flags other than IFF_LOGINT_FLAGS. This
10239  *        is because applications generally does SICGLIFFLAGS and
10240  *        ORs in the new flags (that affects the logical) and does a
10241  *        SIOCSLIFFLAGS. Thus, "flags" below could contain bits other
10242  *        than IFF_LOGINT_FLAGS. One could check whether "turn_on" - the
10243  *        flags that will be turned on is correct with respect to
10244  *        ipif_id 0. For backward compatibility reasons, it is not done.
10245  */
10246 /* ARGSUSED */
10247 int
10248 ip_sioctl_flags(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10249     ip_ioctl_cmd_t *ipip, void *if_req)
10250 {
10251         uint64_t turn_on;
10252         uint64_t turn_off;
10253         int     err = 0;
10254         phyint_t *phyi;
10255         ill_t *ill;
10256         conn_t *connp;
10257         uint64_t intf_flags;
10258         boolean_t phyint_flags_modified = B_FALSE;
10259         uint64_t flags;
10260         struct ifreq *ifr;
10261         struct lifreq *lifr;
10262         boolean_t set_linklocal = B_FALSE;

10264         ip1dbg(("ip_sioctl_flags(%s:%u %p)\n",
10265             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

10267         ASSERT(IAM_WRITER_IPIF(ipif));

10269         ill = ipif->ipif_ill;
10270         phyi = ill->ill_phyint;

10272         if (ipip->ipi_cmd_type == IF_CMD) {
10273                 ifr = (struct ifreq *)if_req;
10274                 flags =  (uint64_t)(ifr->ifr_flags & 0x0000ffff);
10275         } else {
10276                 lifr = (struct lifreq *)if_req;
10277                 flags = lifr->lifr_flags;
10278         }

10280         intf_flags = ipif->ipif_flags | ill->ill_flags | phyi->phyint_flags;

10282         /*
10283          * Have the flags been set correctly until now?
10284          */
10285         ASSERT((phyi->phyint_flags & ~(IFF_PHYINT_FLAGS)) == 0);
10286         ASSERT((ill->ill_flags & ~(IFF_PHYINTINST_FLAGS)) == 0);
10287         ASSERT((ipif->ipif_flags & ~(IFF_LOGINT_FLAGS)) == 0);
10288         /*
10289          * Compare the new flags to the old, and partition
10290          * into those coming on and those going off.
10291          * For the 16 bit command keep the bits above bit 16 unchanged.
```

```
10292          */
10293         if (ipip->ipi_cmd == SIOCSIFFLAGS)
10294                 flags |= intf_flags & ~0xFFFF;

10296         /*
10297          * Explicitly fail attempts to change flags that are always invalid on
10298          * an IPMP meta-interface.
10299          */
10300         if (IS_IPMP(ill) && ((flags ^ intf_flags) & IFF_IPMP_INVALID))
10301                 return (EINVAL);

10303         ip_sioctl_flags_onoff(ipif, flags, &turn_on, &turn_off);
10304         if ((turn_on|turn_off) == 0)
10305                 return (0);     /* No change */

10307         /*
10308          * All test addresses must be IFF_DEPRECATED (to ensure source address
10309          * selection avoids them) -- so force IFF_DEPRECATED on, and do not
10310          * allow it to be turned off.
10311          */
10312         if ((turn_off & (IFF_DEPRECATED|IFF_NOFAILOVER)) == IFF_DEPRECATED &&
10313             (turn_on|intf_flags) & IFF_NOFAILOVER)
10314                 return (EINVAL);

10316         if ((connp = Q_TO_CONN(q)) == NULL)
10317                 return (EINVAL);

10319         /*
10320          * Only vrrp control socket is allowed to change IFF_UP and
10321          * IFF_NOACCEPT flags when IFF_VRRP is set.
10322          */
10323         if ((intf_flags & IFF_VRRP) && ((turn_off | turn_on) & IFF_UP)) {
10324                 if (!connp->conn_isvrrp)
10325                         return (EINVAL);
10326         }

10328         /*
10329          * The IFF_NOACCEPT flag can only be set on an IFF_VRRP IP address by
10330          * VRRP control socket.
10331          */
10332         if ((turn_off | turn_on) & IFF_NOACCEPT) {
10333                 if (!connp->conn_isvrrp || !(intf_flags & IFF_VRRP))
10334                         return (EINVAL);
10335         }

10337         if (turn_on & IFF_NOFAILOVER) {
10338                 turn_on |= IFF_DEPRECATED;
10339                 flags |= IFF_DEPRECATED;
10340         }

10342         /*
10343          * On underlying interfaces, only allow applications to manage test
10344          * addresses -- otherwise, they may get confused when the address
10345          * moves as part of being brought up.  Likewise, prevent an
10346          * application-managed test address from being converted to a data
10347          * address.  To prevent migration of administratively up addresses in
10348          * the kernel, we don't allow them to be converted either.
10349          */
10350         if (IS_UNDER_IPMP(ill)) {
10351                 const uint64_t appflags = IFF_DHCPRUNNING | IFF_ADDRCONF;

10353                 if ((turn_on & appflags) && !(flags & IFF_NOFAILOVER))
10354                         return (EINVAL);

10356                 if ((turn_off & IFF_NOFAILOVER) &&
10357                     (flags & (appflags | IFF_UP | IFF_DUPLICATE)))
```



```
10358                         return (EINVAL);
10359         }

10361         /*
10362          * Only allow IFF_TEMPORARY flag to be set on
10363          * IPv6 interfaces.
10364          */
10365         if ((turn_on & IFF_TEMPORARY) && !(ipif->ipif_isv6))
10366                 return (EINVAL);

10368         /*
10369          * cannot turn off IFF_NOXMIT on  VNI interfaces.
10370          */
10371         if ((turn_off & IFF_NOXMIT) && IS_VNI(ipif->ipif_ill))
10372                 return (EINVAL);

10374         /*
10375          * Don't allow the IFF_ROUTER flag to be turned on on loopback
10376          * interfaces.  It makes no sense in that context.
10377          */
10378         if ((turn_on & IFF_ROUTER) && (phyi->phyint_flags & PHYI_LOOPBACK))
10379                 return (EINVAL);

10381         /*
10382          * For IPv6 ipif_id 0, don't allow the interface to be up without
10383          * a link local address if IFF_NOLOCAL or IFF_ANYCAST are not set.
10384          * If the link local address isn't set, and can be set, it will get
10385          * set later on in this function.
10386          */
10387         if (ipif->ipif_id == 0 && ipif->ipif_isv6 &&
10388             (flags & IFF_UP) && !(flags & (IFF_NOLOCAL|IFF_ANYCAST)) &&
10389             IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6lcl_addr)) {
10390                 if (ipif_cant_setlinklocal(ipif))
10391                         return (EINVAL);
10392                 set_linklocal = B_TRUE;
10393         }

10395         /*
10396          * If we modify physical interface flags, we'll potentially need to
10397          * send up two routing socket messages for the changes (one for the
10398          * IPv4 ill, and another for the IPv6 ill).  Note that here.
10399          */
10400         if ((turn_on|turn_off) & IFF_PHYINT_FLAGS)
10401                 phyint_flags_modified = B_TRUE;

10403         /*
10404          * All functioning PHYI_STANDBY interfaces start life PHYI_INACTIVE
10405          * (otherwise, we'd immediately use them, defeating standby).  Also,
10406          * since PHYI_INACTIVE has a separate meaning when PHYI_STANDBY is not
10407          * set, don't allow PHYI_STANDBY to be set if PHYI_INACTIVE is already
10408          * set, and clear PHYI_INACTIVE if PHYI_STANDBY is being cleared.  We
10409          * also don't allow PHYI_STANDBY if VNI is enabled since its semantics
10410          * will not be honored.
10411          */
10412         if (turn_on & PHYI_STANDBY) {
10413                 /*
10414                  * No need to grab ill_g_usesrc_lock here; see the
10415                  * synchronization notes in ip.c.
10416                  */
10417                 if (ill->ill_usesrc_grp_next != NULL ||
10418                     intf_flags & PHYI_INACTIVE)
10419                         return (EINVAL);
10420                 if (!(flags & PHYI_FAILED)) {
10421                         flags |= PHYI_INACTIVE;
10422                         turn_on |= PHYI_INACTIVE;
10423                 }
```

```
10424         }
10425
10426         if (turn_off & PHYI_STANDBY) {
10427                 flags &= ~PHYI_INACTIVE;
10428                 turn_off |= PHYI_INACTIVE;
10429         }
10430
10431         /*
10432          * PHYI_FAILED and PHYI_INACTIVE are mutually exclusive; fail if both
10433          * would end up on.
10434          */
10435         if ((flags & (PHYI_FAILED | PHYI_INACTIVE)) ==
10436             (PHYI_FAILED | PHYI_INACTIVE))
10437                 return (EINVAL);
10438
10439         /*
10440          * If ILLF_ROUTER changes, we need to change the ip forwarding
10441          * status of the interface.
10442          */
10443         if ((turn_on | turn_off) & ILLF_ROUTER) {
10444                 err = ill_forward_set(ill, ((turn_on & ILLF_ROUTER) != 0));
10445                 if (err != 0)
10446                         return (err);
10447         }
10448
10449         /*
10450          * If the interface is not UP and we are not going to
10451          * bring it UP, record the flags and return. When the
10452          * interface comes UP later, the right actions will be
10453          * taken.
10454          */
10455         if (!(ipif->ipif_flags & IPIF_UP) &&
10456             !(turn_on & IPIF_UP)) {
10457                 /* Record new flags in their respective places. */
10458                 mutex_enter(&ill->ill_lock);
10459                 mutex_enter(&ill->ill_phyint->phyint_lock);
10460                 ipif->ipif_flags |= (turn_on & IFF_LOGINT_FLAGS);
10461                 ipif->ipif_flags &= (~turn_off & IFF_LOGINT_FLAGS);
10462                 ill->ill_flags |= (turn_on & IFF_PHYINTINST_FLAGS);
10463                 ill->ill_flags &= (~turn_off & IFF_PHYINTINST_FLAGS);
10464                 phyi->phyint_flags |= (turn_on & IFF_PHYINT_FLAGS);
10465                 phyi->phyint_flags &= (~turn_off & IFF_PHYINT_FLAGS);
10466                 mutex_exit(&ill->ill_lock);
10467                 mutex_exit(&ill->ill_phyint->phyint_lock);
10468
10469                 /*
10470                  * PHYI_FAILED, PHYI_INACTIVE, and PHYI_OFFLINE are all the
10471                  * same to the kernel: if any of them has been set by
10472                  * userland, the interface cannot be used for data traffic.
10473                  */
10474                 if ((turn_on|turn_off) &
10475                     (PHYI_FAILED | PHYI_INACTIVE | PHYI_OFFLINE)) {
10476                         ASSERT(!IS_IPMP(ill));
10477                         /*
10478                          * It's possible the ill is part of an "anonymous"
10479                          * IPMP group rather than a real group.  In that case,
10480                          * there are no other interfaces in the group and thus
10481                          * no need to call ipmp_phyint_refresh_active().
10482                          */
10483                         if (IS_UNDER_IPMP(ill))
10484                                 ipmp_phyint_refresh_active(phyi);
10485                 }
10486
10487                 if (phyint_flags_modified) {
10488                         if (phyi->phyint_illv4 != NULL) {
10489                                 ip_rts_ifmsg(phyi->phyint_illv4->
```



```
10490                                     ill_ipif, RTSQ_DEFAULT);
10491                         }
10492                         if (phyi->phyint_illv6 != NULL) {
10493                                 ip_rts_ifmsg(phyi->phyint_illv6->
10494                                     ill_ipif, RTSQ_DEFAULT);
10495                         }
10496                 }
10497                 /* The default multicast interface might have changed */
10498                 ire_increment_multicast_generation(ill->ill_ipst,
10499                     ill->ill_isv6);
10500
10501                 return (0);
10502         } else if (set_linklocal) {
10503                 mutex_enter(&ill->ill_lock);
10504                 if (set_linklocal)
10505                         ipif->ipif_state_flags |= IPIF_SET_LINKLOCAL;
10506                 mutex_exit(&ill->ill_lock);
10507         }
10508
10509         /*
10510          * Disallow IPv6 interfaces coming up that have the unspecified address,
10511          * or point-to-point interfaces with an unspecified destination. We do
10512          * allow the address to be unspecified for IPIF_NOLOCAL interfaces that
10513          * have a subnet assigned, which is how in.ndpd currently manages its
10514          * onlink prefix list when no addresses are configured with those
10515          * prefixes.
10516          */
10517         if (ipif->ipif_isv6 &&
10518             ((IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6lcl_addr) &&
10519             (!(ipif->ipif_flags & IPIF_NOLOCAL) && !(turn_on & IPIF_NOLOCAL) ||
10520             IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6subnet))) ||
10521             ((ipif->ipif_flags & IPIF_POINTOPOINT) &&
10522             IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6pp_dst_addr)))) {
10523                 return (EINVAL);
10524         }
10525
10526         /*
10527          * Prevent IPv4 point-to-point interfaces with a 0.0.0.0 destination
10528          * from being brought up.
10529          */
10530         if (!ipif->ipif_isv6 &&
10531             ((ipif->ipif_flags & IPIF_POINTOPOINT) &&
10532             ipif->ipif_pp_dst_addr == INADDR_ANY)) {
10533                 return (EINVAL);
10534         }
10535
10536         /*
10537          * If we are going to change one or more of the flags that are
10538          * IPIF_UP, IPIF_DEPRECATED, IPIF_NOXMIT, IPIF_NOLOCAL, ILLF_NOARP,
10539          * ILLF_NONUD, IPIF_PRIVATE, IPIF_ANYCAST, IPIF_PREFERRED, and
10540          * IPIF_NOFAILOVER, we will take special action.  This is
10541          * done by bring the ipif down, changing the flags and bringing
10542          * it back up again.  For IPIF_NOFAILOVER, the act of bringing it
10543          * back up will trigger the address to be moved.
10544          *
10545          * If we are going to change IFF_NOACCEPT, we need to bring
10546          * all the ipifs down then bring them up again.  The act of
10547          * bringing all the ipifs back up will trigger the local
10548          * ires being recreated with "no_accept" set/cleared.
10549          *
10550          * Note that ILLF_NOACCEPT is always set separately from the
10551          * other flags.
10552          */
10553         if ((turn_on|turn_off) &
10554             (IPIF_UP|IPIF_DEPRECATED|IPIF_NOXMIT|IPIF_NOLOCAL|ILLF_NOARP|
10555             ILLF_NONUD|IPIF_PRIVATE|IPIF_ANYCAST|IPIF_PREFERRED|
```

```
10556             IPIF_NOFAILOVER)) {
10557                 /*
10558                  * ipif_down() will ire_delete bcast ire's for the subnet,
10559                  * while the ire_identical_ref tracks the case of IRE_BROADCAST
10560                  * entries shared between multiple ipifs on the same subnet.
10561                  */
10562                 if (((ipif->ipif_flags | turn_on) & IPIF_UP) &&
10563                     !(turn_off & IPIF_UP)) {
10564                         if (ipif->ipif_flags & IPIF_UP)
10565                                 ill->ill_logical_down = 1;
10566                         turn_on &= ~IPIF_UP;
10567                 }
10568                 err = ipif_down(ipif, q, mp);
10569                 ip1dbg(("ipif_down returns %d err ", err));
10570                 if (err == EINPROGRESS)
10571                         return (err);
10572                 (void) ipif_down_tail(ipif);
10573         } else if ((turn_on|turn_off) & ILLF_NOACCEPT) {
10574                 /*
10575                  * If we can quiesce the ill, then continue.  If not, then
10576                  * ip_sioctl_flags_tail() will be called from
10577                  * ipif_ill_refrele_tail().
10578                  */
10579                 ill_down_ipifs(ill, B_TRUE);

10581                 mutex_enter(&connp->conn_lock);
10582                 mutex_enter(&ill->ill_lock);
10583                 if (!ill_is_quiescent(ill)) {
10584                         boolean_t success;

10586                         success = ipsq_pending_mp_add(connp, ill->ill_ipif,
10587                             q, mp, ILL_DOWN);
10588                         mutex_exit(&ill->ill_lock);
10589                         mutex_exit(&connp->conn_lock);
10590                         return (success ? EINPROGRESS : EINTR);
10591                 }
10592                 mutex_exit(&ill->ill_lock);
10593                 mutex_exit(&connp->conn_lock);
10594         }
10595         return (ip_sioctl_flags_tail(ipif, flags, q, mp));
10596 }

10598 static int
10599 ip_sioctl_flags_tail(ipif_t *ipif, uint64_t flags, queue_t *q, mblk_t *mp)
10600 {
10601         ill_t   *ill;
10602         phyint_t *phyi;
10603         uint64_t turn_on, turn_off;
10604         boolean_t phyint_flags_modified = B_FALSE;
10605         int     err = 0;
10606         boolean_t set_linklocal = B_FALSE;

10608         ip1dbg(("ip_sioctl_flags_tail(%s:%u)\n",
10609             ipif->ipif_ill->ill_name, ipif->ipif_id));

10611         ASSERT(IAM_WRITER_IPIF(ipif));

10613         ill = ipif->ipif_ill;
10614         phyi = ill->ill_phyint;

10616         ip_sioctl_flags_onoff(ipif, flags, &turn_on, &turn_off);

10618         /*
10619          * IFF_UP is handled separately.
10620          */
10621         turn_on &= ~IFF_UP;
```



```
10622         turn_off &= ~IFF_UP;

10624         if ((turn_on|turn_off) & IFF_PHYINT_FLAGS)
10625                 phyint_flags_modified = B_TRUE;

10627         /*
10628          * Now we change the flags. Track current value of
10629          * other flags in their respective places.
10630          */
10631         mutex_enter(&ill->ill_lock);
10632         mutex_enter(&phyi->phyint_lock);
10633         ipif->ipif_flags |= (turn_on & IFF_LOGINT_FLAGS);
10634         ipif->ipif_flags &= (~turn_off & IFF_LOGINT_FLAGS);
10635         ill->ill_flags |= (turn_on & IFF_PHYINTINST_FLAGS);
10636         ill->ill_flags &= (~turn_off & IFF_PHYINTINST_FLAGS);
10637         phyi->phyint_flags |= (turn_on & IFF_PHYINT_FLAGS);
10638         phyi->phyint_flags &= (~turn_off & IFF_PHYINT_FLAGS);
10639         if (ipif->ipif_state_flags & IPIF_SET_LINKLOCAL) {
10640                 set_linklocal = B_TRUE;
10641                 ipif->ipif_state_flags &= ~IPIF_SET_LINKLOCAL;
10642         }

10644         mutex_exit(&ill->ill_lock);
10645         mutex_exit(&phyi->phyint_lock);

10647         if (set_linklocal)
10648                 (void) ipif_setlinklocal(ipif);

10650         /*
10651          * PHYI_FAILED, PHYI_INACTIVE, and PHYI_OFFLINE are all the same to
10652          * the kernel: if any of them has been set by userland, the interface
10653          * cannot be used for data traffic.
10654          */
10655         if ((turn_on|turn_off) & (PHYI_FAILED | PHYI_INACTIVE | PHYI_OFFLINE)) {
10656                 ASSERT(!IS_IPMP(ill));
10657                 /*
10658                  * It's possible the ill is part of an "anonymous" IPMP group
10659                  * rather than a real group.  In that case, there are no other
10660                  * interfaces in the group and thus no need for us to call
10661                  * ipmp_phyint_refresh_active().
10662                  */
10663                 if (IS_UNDER_IPMP(ill))
10664                         ipmp_phyint_refresh_active(phyi);
10665         }

10667         if ((turn_on|turn_off) & ILLF_NOACCEPT) {
10668                 /*
10669                  * If the ILLF_NOACCEPT flag is changed, bring up all the
10670                  * ipifs that were brought down.
10671                  *
10672                  * The routing sockets messages are sent as the result
10673                  * of ill_up_ipifs(), further, SCTP's IPIF list was updated
10674                  * as well.
10675                  */
10676                 err = ill_up_ipifs(ill, q, mp);
10677         } else if ((flags & IFF_UP) && !(ipif->ipif_flags & IPIF_UP)) {
10678                 /*
10679                  * XXX ipif_up really does not know whether a phyint flags
10680                  * was modified or not. So, it sends up information on
10681                  * only one routing sockets message. As we don't bring up
10682                  * the interface and also set PHYI_ flags simultaneously
10683                  * it should be okay.
10684                  */
10685                 err = ipif_up(ipif, q, mp);
10686         } else {
10687                 /*
```

```
10688                  * Make sure routing socket sees all changes to the flags.
10689                  * ipif_up_done* handles this when we use ipif_up.
10690                  */
10691                 if (phyint_flags_modified) {
10692                         if (phyi->phyint_illv4 != NULL) {
10693                                 ip_rts_ifmsg(phyi->phyint_illv4->
10694                                     ill_ipif, RTSQ_DEFAULT);
10695                         }
10696                         if (phyi->phyint_illv6 != NULL) {
10697                                 ip_rts_ifmsg(phyi->phyint_illv6->
10698                                     ill_ipif, RTSQ_DEFAULT);
10699                         }
10700                 } else {
10701                         ip_rts_ifmsg(ipif, RTSQ_DEFAULT);
10702                 }
10703                 /*
10704                  * Update the flags in SCTP's IPIF list, ipif_up() will do
10705                  * this in need_up case.
10706                  */
10707                 sctp_update_ipif(ipif, SCTP_IPIF_UPDATE);
10708         }

10710         /* The default multicast interface might have changed */
10711         ire_increment_multicast_generation(ill->ill_ipst, ill->ill_isv6);
10712         return (err);
10713 }

10715 /*
10716  * Restart the flags operation now that the refcounts have dropped to zero.
10717  */
10718 /* ARGSUSED */
10719 int
10720 ip_sioctl_flags_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10721     ip_ioctl_cmd_t *ipip, void *if_req)
10722 {
10723         uint64_t flags;
10724         struct ifreq *ifr = if_req;
10725         struct lifreq *lifr = if_req;
10726         uint64_t turn_on, turn_off;

10728         ip1dbg(("ip_sioctl_flags_restart(%s:%u %p)\n",
10729             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

10731         if (ipip->ipi_cmd_type == IF_CMD) {
10732                 /* cast to uint16_t prevents unwanted sign extension */
10733                 flags = (uint16_t)ifr->ifr_flags;
10734         } else {
10735                 flags = lifr->lifr_flags;
10736         }

10738         /*
10739          * If this function call is a result of the ILLF_NOACCEPT flag
10740          * change, do not call ipif_down_tail(). See ip_sioctl_flags().
10741          */
10742         ip_sioctl_flags_onoff(ipif, flags, &turn_on, &turn_off);
10743         if (!((turn_on|turn_off) & ILLF_NOACCEPT))
10744                 (void) ipif_down_tail(ipif);

10746         return (ip_sioctl_flags_tail(ipif, flags, q, mp));
10747 }

10749 /*
10750  * Can operate on either a module or a driver queue.
10751  */
10752 /* ARGSUSED */
10753 int
```



```
10754 ip_sioctl_get_flags(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10755     ip_ioctl_cmd_t *ipip, void *if_req)
10756 {
10757         /*
10758          * Has the flags been set correctly till now ?
10759          */
10760         ill_t *ill = ipif->ipif_ill;
10761         phyint_t *phyi = ill->ill_phyint;

10763         ip1dbg(("ip_sioctl_get_flags(%s:%u %p)\n",
10764             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10765         ASSERT((phyi->phyint_flags & ~(IFF_PHYINT_FLAGS)) == 0);
10766         ASSERT((ill->ill_flags & ~(IFF_PHYINTINST_FLAGS)) == 0);
10767         ASSERT((ipif->ipif_flags & ~(IFF_LOGINT_FLAGS)) == 0);

10769         /*
10770          * Need a lock since some flags can be set even when there are
10771          * references to the ipif.
10772          */
10773         mutex_enter(&ill->ill_lock);
10774         if (ipip->ipi_cmd_type == IF_CMD) {
10775                 struct ifreq *ifr = (struct ifreq *)if_req;

10777                 /* Get interface flags (low 16 only). */
10778                 ifr->ifr_flags = ((ipif->ipif_flags |
10779                     ill->ill_flags | phyi->phyint_flags) & 0xffff);
10780         } else {
10781                 struct lifreq *lifr = (struct lifreq *)if_req;

10783                 /* Get interface flags. */
10784                 lifr->lifr_flags = ipif->ipif_flags |
10785                     ill->ill_flags | phyi->phyint_flags;
10786         }
10787         mutex_exit(&ill->ill_lock);
10788         return (0);
10789 }

10791 /*
10792  * We allow the MTU to be set on an ILL, but not have it be different
10793  * for different IPIFs since we don't actually send packets on IPIFs.
10794  */
10795 /* ARGSUSED */
10796 int
10797 ip_sioctl_mtu(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10798     ip_ioctl_cmd_t *ipip, void *if_req)
10799 {
10800         int mtu;
10801         int ip_min_mtu;
10802         struct ifreq    *ifr;
10803         struct lifreq *lifr;
10804         ill_t   *ill;

10806         ip1dbg(("ip_sioctl_mtu(%s:%u %p)\n", ipif->ipif_ill->ill_name,
10807             ipif->ipif_id, (void *)ipif));
10808         if (ipip->ipi_cmd_type == IF_CMD) {
10809                 ifr = (struct ifreq *)if_req;
10810                 mtu = ifr->ifr_metric;
10811         } else {
10812                 lifr = (struct lifreq *)if_req;
10813                 mtu = lifr->lifr_mtu;
10814         }
10815         /* Only allow for logical unit zero i.e. not on "bge0:17" */
10816         if (ipif->ipif_id != 0)
10817                 return (EINVAL);

10819         ill = ipif->ipif_ill;
```

```
10820         if (ipif->ipif_isv6)
10821                 ip_min_mtu = IPV6_MIN_MTU;
10822         else
10823                 ip_min_mtu = IP_MIN_MTU;

10825         mutex_enter(&ill->ill_lock);
10826         if (mtu > ill->ill_max_frag || mtu < ip_min_mtu) {
10827                 mutex_exit(&ill->ill_lock);
10828                 return (EINVAL);
10829         }
10830         /* Avoid increasing ill_mc_mtu */
10831         if (ill->ill_mc_mtu > mtu)
10832                 ill->ill_mc_mtu = mtu;

10834         /*
10835          * The dce and fragmentation code can handle changes to ill_mtu
10836          * concurrent with sending/fragmenting packets.
10837          */
10838         ill->ill_mtu = mtu;
10839         ill->ill_flags |= ILLF_FIXEDMTU;
10840         mutex_exit(&ill->ill_lock);

10842         /*
10843          * Make sure all dce_generation checks find out
10844          * that ill_mtu/ill_mc_mtu has changed.
10845          */
10846         dce_increment_all_generations(ill->ill_isv6, ill->ill_ipst);

10848         /*
10849          * Refresh IPMP meta-interface MTU if necessary.
10850          */
10851         if (IS_UNDER_IPMP(ill))
10852                 ipmp_illgrp_refresh_mtu(ill->ill_grp);

10854         /* Update the MTU in SCTP's list */
10855         sctp_update_ipif(ipif, SCTP_IPIF_UPDATE);
10856         return (0);
10857 }

10859 /* Get interface MTU. */
10860 /* ARGSUSED */
10861 int
10862 ip_sioctl_get_mtu(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10863         ip_ioctl_cmd_t *ipip, void *if_req)
10864 {
10865         struct ifreq    *ifr;
10866         struct lifreq   *lifr;

10868         ip1dbg(("ip_sioctl_get_mtu(%s:%u %p)\n",
10869             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

10871         /*
10872          * We allow a get on any logical interface even though the set
10873          * can only be done on logical unit 0.
10874          */
10875         if (ipip->ipi_cmd_type == IF_CMD) {
10876                 ifr = (struct ifreq *)if_req;
10877                 ifr->ifr_metric = ipif->ipif_ill->ill_mtu;
10878         } else {
10879                 lifr = (struct lifreq *)if_req;
10880                 lifr->lifr_mtu = ipif->ipif_ill->ill_mtu;
10881         }
10882         return (0);
10883 }

10885 /* Set interface broadcast address. */
```



```
10886 /* ARGSUSED2 */
10887 int
10888 ip_sioctl_brdaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10889         ip_ioctl_cmd_t *ipip, void *if_req)
10890 {
10891         ipaddr_t addr;
10892         ire_t   *ire;
10893         ill_t           *ill = ipif->ipif_ill;
10894         ip_stack_t      *ipst = ill->ill_ipst;

10896         ip1dbg(("ip_sioctl_brdaddr(%s:%u)\n", ill->ill_name,
10897             ipif->ipif_id));

10899         ASSERT(IAM_WRITER_IPIF(ipif));
10900         if (!(ipif->ipif_flags & IPIF_BROADCAST))
10901                 return (EADDRNOTAVAIL);

10903         ASSERT(!(ipif->ipif_isv6));     /* No IPv6 broadcast */

10905         if (sin->sin_family != AF_INET)
10906                 return (EAFNOSUPPORT);

10908         addr = sin->sin_addr.s_addr;

10910         if (ipif->ipif_flags & IPIF_UP) {
10911                 /*
10912                  * If we are already up, make sure the new
10913                  * broadcast address makes sense.  If it does,
10914                  * there should be an IRE for it already.
10915                  */
10916                 ire = ire_ftable_lookup_v4(addr, 0, 0, IRE_BROADCAST,
10917                     ill, ipif->ipif_zoneid, NULL,
10918                     (MATCH_IRE_ILL | MATCH_IRE_TYPE), 0, ipst, NULL);
10919                 if (ire == NULL) {
10920                         return (EINVAL);
10921                 } else {
10922                         ire_refrele(ire);
10923                 }
10924         }
10925         /*
10926          * Changing the broadcast addr for this ipif. Since the IRE_BROADCAST
10927          * needs to already exist we never need to change the set of
10928          * IRE_BROADCASTs when we are UP.
10929          */
10930         if (addr != ipif->ipif_brd_addr)
10931                 IN6_IPADDR_TO_V4MAPPED(addr, &ipif->ipif_v6brd_addr);

10933         return (0);
10934 }

10936 /* Get interface broadcast address. */
10937 /* ARGSUSED */
10938 int
10939 ip_sioctl_get_brdaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10940     ip_ioctl_cmd_t *ipip, void *if_req)
10941 {
10942         ip1dbg(("ip_sioctl_get_brdaddr(%s:%u %p)\n",
10943             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10944         if (!(ipif->ipif_flags & IPIF_BROADCAST))
10945                 return (EADDRNOTAVAIL);

10947         /* IPIF_BROADCAST not possible with IPv6 */
10948         ASSERT(!ipif->ipif_isv6);
10949         *sin = sin_null;
10950         sin->sin_family = AF_INET;
10951         sin->sin_addr.s_addr = ipif->ipif_brd_addr;
```

```
10952         return (0);
10953 }

10955 /*
10956  * This routine is called to handle the SIOCS*IFNETMASK IOCTL.
10957  */
10958 /* ARGSUSED */
10959 int
10960 ip_sioctl_netmask(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10961     ip_ioctl_cmd_t *ipip, void *if_req)
10962 {
10963         int err = 0;
10964         in6_addr_t v6mask;

10966         ip1dbg(("ip_sioctl_netmask(%s:%u %p)\n",
10967             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

10969         ASSERT(IAM_WRITER_IPIF(ipif));

10971         if (ipif->ipif_isv6) {
10972                 sin6_t *sin6;

10974                 if (sin->sin_family != AF_INET6)
10975                         return (EAFNOSUPPORT);

10977                 sin6 = (sin6_t *)sin;
10978                 v6mask = sin6->sin6_addr;
10979         } else {
10980                 ipaddr_t mask;

10982                 if (sin->sin_family != AF_INET)
10983                         return (EAFNOSUPPORT);

10985                 mask = sin->sin_addr.s_addr;
10986                 if (!ip_contiguous_mask(ntohl(mask)))
10987                         return (ENOTSUP);
10988                 V4MASK_TO_V6(mask, v6mask);
10989         }

10991         /*
10992          * No big deal if the interface isn't already up, or the mask
10993          * isn't really changing, or this is pt-pt.
10994          */
10995         if (!(ipif->ipif_flags & IPIF_UP) ||
10996             IN6_ARE_ADDR_EQUAL(&v6mask, &ipif->ipif_v6net_mask) ||
10997             (ipif->ipif_flags & IPIF_POINTOPOINT)) {
10998                 ipif->ipif_v6net_mask = v6mask;
10999                 if ((ipif->ipif_flags & IPIF_POINTOPOINT) == 0) {
11000                         V6_MASK_COPY(ipif->ipif_v6lcl_addr,
11001                             ipif->ipif_v6net_mask,
11002                             ipif->ipif_v6subnet);
11003                 }
11004                 return (0);
11005         }
11006         /*
11007          * Make sure we have valid net and subnet broadcast ire's
11008          * for the old netmask, if needed by other logical interfaces.
11009          */
11010         err = ipif_logical_down(ipif, q, mp);
11011         if (err == EINPROGRESS)
11012                 return (err);
11013         (void) ipif_down_tail(ipif);
11014         err = ip_sioctl_netmask_tail(ipif, sin, q, mp);
11015         return (err);
11016 }
```



```
11018 static int
11019 ip_sioctl_netmask_tail(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp)
11020 {
11021         in6_addr_t v6mask;
11022         int err = 0;

11024         ip1dbg(("ip_sioctl_netmask_tail(%s:%u %p)\n",
11025             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

11027         if (ipif->ipif_isv6) {
11028                 sin6_t *sin6;

11030                 sin6 = (sin6_t *)sin;
11031                 v6mask = sin6->sin6_addr;
11032         } else {
11033                 ipaddr_t mask;

11035                 mask = sin->sin_addr.s_addr;
11036                 V4MASK_TO_V6(mask, v6mask);
11037         }

11039         ipif->ipif_v6net_mask = v6mask;
11040         if ((ipif->ipif_flags & IPIF_POINTOPOINT) == 0) {
11041                 V6_MASK_COPY(ipif->ipif_v6lcl_addr, ipif->ipif_v6net_mask,
11042                     ipif->ipif_v6subnet);
11043         }
11044         err = ipif_up(ipif, q, mp);

11046         if (err == 0 || err == EINPROGRESS) {
11047                 /*
11048                  * The interface must be DL_BOUND if this packet has to
11049                  * go out on the wire. Since we only go through a logical
11050                  * down and are bound with the driver during an internal
11051                  * down/up that is satisfied.
11052                  */
11053                 if (!ipif->ipif_isv6 && ipif->ipif_ill->ill_wq != NULL) {
11054                         /* Potentially broadcast an address mask reply. */
11055                         ipif_mask_reply(ipif);
11056                 }
11057         }
11058         return (err);
11059 }

11061 /* ARGSUSED */
11062 int
11063 ip_sioctl_netmask_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11064     ip_ioctl_cmd_t *ipip, void *if_req)
11065 {
11066         ip1dbg(("ip_sioctl_netmask_restart(%s:%u %p)\n",
11067             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11068         (void) ipif_down_tail(ipif);
11069         return (ip_sioctl_netmask_tail(ipif, sin, q, mp));
11070 }

11072 /* Get interface net mask. */
11073 /* ARGSUSED */
11074 int
11075 ip_sioctl_get_netmask(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11076     ip_ioctl_cmd_t *ipip, void *if_req)
11077 {
11078         struct lifreq *lifr = (struct lifreq *)if_req;
11079         struct sockaddr_in6 *sin6 = (sin6_t *)sin;

11081         ip1dbg(("ip_sioctl_get_netmask(%s:%u %p)\n",
11082             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
```

```
11084         /*
11085          * net mask can't change since we have a reference to the ipif.
11086          */
11087         if (ipif->ipif_isv6) {
11088                 ASSERT(ipip->ipi_cmd_type == LIF_CMD);
11089                 *sin6 = sin6_null;
11090                 sin6->sin6_family = AF_INET6;
11091                 sin6->sin6_addr = ipif->ipif_v6net_mask;
11092                 lifr->lifr_addrlen =
11093                     ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
11094         } else {
11095                 *sin = sin_null;
11096                 sin->sin_family = AF_INET;
11097                 sin->sin_addr.s_addr = ipif->ipif_net_mask;
11098                 if (ipip->ipi_cmd_type == LIF_CMD) {
11099                         lifr->lifr_addrlen =
11100                             ip_mask_to_plen(ipif->ipif_net_mask);
11101                 }
11102         }
11103         return (0);
11104 }

11106 /* ARGSUSED */
11107 int
11108 ip_sioctl_metric(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11109     ip_ioctl_cmd_t *ipip, void *if_req)
11110 {
11111         ip1dbg(("ip_sioctl_metric(%s:%u %p)\n",
11112             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

11114         /*
11115          * Since no applications should ever be setting metrics on underlying
11116          * interfaces, we explicitly fail to smoke 'em out.
11117          */
11118         if (IS_UNDER_IPMP(ipif->ipif_ill))
11119                 return (EINVAL);

11121         /*
11122          * Set interface metric.  We don't use this for
11123          * anything but we keep track of it in case it is
11124          * important to routing applications or such.
11125          */
11126         if (ipip->ipi_cmd_type == IF_CMD) {
11127                 struct ifreq    *ifr;

11129                 ifr = (struct ifreq *)if_req;
11130                 ipif->ipif_ill->ill_metric = ifr->ifr_metric;
11131         } else {
11132                 struct lifreq   *lifr;

11134                 lifr = (struct lifreq *)if_req;
11135                 ipif->ipif_ill->ill_metric = lifr->lifr_metric;
11136         }
11137         return (0);
11138 }

11140 /* ARGSUSED */
11141 int
11142 ip_sioctl_get_metric(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11143     ip_ioctl_cmd_t *ipip, void *if_req)
11144 {
11145         /* Get interface metric. */
11146         ip1dbg(("ip_sioctl_get_metric(%s:%u %p)\n",
11147             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

11149         if (ipip->ipi_cmd_type == IF_CMD) {
```



```
11150                 struct ifreq    *ifr;

11152                 ifr = (struct ifreq *)if_req;
11153                 ifr->ifr_metric = ipif->ipif_ill->ill_metric;
11154         } else {
11155                 struct lifreq   *lifr;

11157                 lifr = (struct lifreq *)if_req;
11158                 lifr->lifr_metric = ipif->ipif_ill->ill_metric;
11159         }

11161         return (0);
11162 }

11164 /* ARGSUSED */
11165 int
11166 ip_sioctl_muxid(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11167     ip_ioctl_cmd_t *ipip, void *if_req)
11168 {
11169         int     arp_muxid;

11171         ip1dbg(("ip_sioctl_muxid(%s:%u %p)\n",
11172             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11173         /*
11174          * Set the muxid returned from I_PLINK.
11175          */
11176         if (ipip->ipi_cmd_type == IF_CMD) {
11177                 struct ifreq *ifr = (struct ifreq *)if_req;

11179                 ipif->ipif_ill->ill_muxid = ifr->ifr_ip_muxid;
11180                 arp_muxid = ifr->ifr_arp_muxid;
11181         } else {
11182                 struct lifreq *lifr = (struct lifreq *)if_req;

11184                 ipif->ipif_ill->ill_muxid = lifr->lifr_ip_muxid;
11185                 arp_muxid = lifr->lifr_arp_muxid;
11186         }
11187         arl_set_muxid(ipif->ipif_ill, arp_muxid);
11188         return (0);
11189 }

11191 /* ARGSUSED */
11192 int
11193 ip_sioctl_get_muxid(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11194     ip_ioctl_cmd_t *ipip, void *if_req)
11195 {
11196         int     arp_muxid = 0;

11198         ip1dbg(("ip_sioctl_get_muxid(%s:%u %p)\n",
11199             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11200         /*
11201          * Get the muxid saved in ill for I_PUNLINK.
11202          */
11203         arp_muxid = arl_get_muxid(ipif->ipif_ill);
11204         if (ipip->ipi_cmd_type == IF_CMD) {
11205                 struct ifreq *ifr = (struct ifreq *)if_req;

11207                 ifr->ifr_ip_muxid = ipif->ipif_ill->ill_muxid;
11208                 ifr->ifr_arp_muxid = arp_muxid;
11209         } else {
11210                 struct lifreq *lifr = (struct lifreq *)if_req;

11212                 lifr->lifr_ip_muxid = ipif->ipif_ill->ill_muxid;
11213                 lifr->lifr_arp_muxid = arp_muxid;
11214         }
11215         return (0);
```

```
11216 }

11218 /*
11219  * Set the subnet prefix. Does not modify the broadcast address.
11220  */
11221 /* ARGSUSED */
11222 int
11223 ip_sioctl_subnet(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11224     ip_ioctl_cmd_t *ipip, void *if_req)
11225 {
11226         int err = 0;
11227         in6_addr_t v6addr;
11228         in6_addr_t v6mask;
11229         boolean_t need_up = B_FALSE;
11230         int addrlen;

11232         ip1dbg(("ip_sioctl_subnet(%s:%u %p)\n",
11233             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

11235         ASSERT(IAM_WRITER_IPIF(ipif));
11236         addrlen = ((struct lifreq *)if_req)->lifr_addrlen;

11238         if (ipif->ipif_isv6) {
11239                 sin6_t *sin6;

11241                 if (sin->sin_family != AF_INET6)
11242                         return (EAFNOSUPPORT);

11244                 sin6 = (sin6_t *)sin;
11245                 v6addr = sin6->sin6_addr;
11246                 if (!ip_remote_addr_ok_v6(&v6addr, &ipv6_all_ones))
11247                         return (EADDRNOTAVAIL);
11248         } else {
11249                 ipaddr_t addr;

11251                 if (sin->sin_family != AF_INET)
11252                         return (EAFNOSUPPORT);

11254                 addr = sin->sin_addr.s_addr;
11255                 if (!ip_addr_ok_v4(addr, 0xFFFFFFFF))
11256                         return (EADDRNOTAVAIL);
11257                 IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
11258                 /* Add 96 bits */
11259                 addrlen += IPV6_ABITS - IP_ABITS;
11260         }

11262         if (ip_plen_to_mask_v6(addrlen, &v6mask) == NULL)
11263                 return (EINVAL);

11265         /* Check if bits in the address is set past the mask */
11266         if (!V6_MASK_EQ(v6addr, v6mask, v6addr))
11267                 return (EINVAL);

11269         if (IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6subnet, &v6addr) &&
11270             IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6net_mask, &v6mask))
11271                 return (0);     /* No change */

11273         if (ipif->ipif_flags & IPIF_UP) {
11274                 /*
11275                  * If the interface is already marked up,
11276                  * we call ipif_down which will take care
11277                  * of ditching any IREs that have been set
11278                  * up based on the old interface address.
11279                  */
11280                 err = ipif_logical_down(ipif, q, mp);
11281                 if (err == EINPROGRESS)
```



```
11282                         return (err);
11283                 (void) ipif_down_tail(ipif);
11284                 need_up = B_TRUE;
11285         }

11287         err = ip_sioctl_subnet_tail(ipif, v6addr, v6mask, q, mp, need_up);
11288         return (err);
11289 }

11291 static int
11292 ip_sioctl_subnet_tail(ipif_t *ipif, in6_addr_t v6addr, in6_addr_t v6mask,
11293     queue_t *q, mblk_t *mp, boolean_t need_up)
11294 {
11295         ill_t   *ill = ipif->ipif_ill;
11296         int     err = 0;

11298         ip1dbg(("ip_sioctl_subnet_tail(%s:%u %p)\n",
11299             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

11301         /* Set the new address. */
11302         mutex_enter(&ill->ill_lock);
11303         ipif->ipif_v6net_mask = v6mask;
11304         if ((ipif->ipif_flags & IPIF_POINTOPOINT) == 0) {
11305                 V6_MASK_COPY(v6addr, ipif->ipif_v6net_mask,
11306                     ipif->ipif_v6subnet);
11307         }
11308         mutex_exit(&ill->ill_lock);

11310         if (need_up) {
11311                 /*
11312                  * Now bring the interface back up.  If this
11313                  * is the only IPIF for the ILL, ipif_up
11314                  * will have to re-bind to the device, so
11315                  * we may get back EINPROGRESS, in which
11316                  * case, this IOCTL will get completed in
11317                  * ip_rput_dlpi when we see the DL_BIND_ACK.
11318                  */
11319                 err = ipif_up(ipif, q, mp);
11320                 if (err == EINPROGRESS)
11321                         return (err);
11322         }
11323         return (err);
11324 }

11326 /* ARGSUSED */
11327 int
11328 ip_sioctl_subnet_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11329     ip_ioctl_cmd_t *ipip, void *if_req)
11330 {
11331         int     addrlen;
11332         in6_addr_t v6addr;
11333         in6_addr_t v6mask;
11334         struct lifreq *lifr = (struct lifreq *)if_req;

11336         ip1dbg(("ip_sioctl_subnet_restart(%s:%u %p)\n",
11337             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11338         (void) ipif_down_tail(ipif);

11340         addrlen = lifr->lifr_addrlen;
11341         if (ipif->ipif_isv6) {
11342                 sin6_t *sin6;

11344                 sin6 = (sin6_t *)sin;
11345                 v6addr = sin6->sin6_addr;
11346         } else {
11347                 ipaddr_t addr;
```

```
11349                 addr = sin->sin_addr.s_addr;
11350                 IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
11351                 addrlen += IPV6_ABITS - IP_ABITS;
11352         }
11353         (void) ip_plen_to_mask_v6(addrlen, &v6mask);

11355         return (ip_sioctl_subnet_tail(ipif, v6addr, v6mask, q, mp, B_TRUE));
11356 }

11358 /* ARGSUSED */
11359 int
11360 ip_sioctl_get_subnet(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11361     ip_ioctl_cmd_t *ipip, void *if_req)
11362 {
11363         struct lifreq *lifr = (struct lifreq *)if_req;
11364         struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *)sin;

11366         ip1dbg(("ip_sioctl_get_subnet(%s:%u %p)\n",
11367             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11368         ASSERT(ipip->ipi_cmd_type == LIF_CMD);

11370         if (ipif->ipif_isv6) {
11371                 *sin6 = sin6_null;
11372                 sin6->sin6_family = AF_INET6;
11373                 sin6->sin6_addr = ipif->ipif_v6subnet;
11374                 lifr->lifr_addrlen =
11375                     ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
11376         } else {
11377                 *sin = sin_null;
11378                 sin->sin_family = AF_INET;
11379                 sin->sin_addr.s_addr = ipif->ipif_subnet;
11380                 lifr->lifr_addrlen = ip_mask_to_plen(ipif->ipif_net_mask);
11381         }
11382         return (0);
11383 }

11385 /*
11386  * Set the IPv6 address token.
11387  */
11388 /* ARGSUSED */
11389 int
11390 ip_sioctl_token(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11391     ip_ioctl_cmd_t *ipi, void *if_req)
11392 {
11393         ill_t *ill = ipif->ipif_ill;
11394         int err;
11395         in6_addr_t v6addr;
11396         in6_addr_t v6mask;
11397         boolean_t need_up = B_FALSE;
11398         int i;
11399         sin6_t *sin6 = (sin6_t *)sin;
11400         struct lifreq *lifr = (struct lifreq *)if_req;
11401         int addrlen;

11403         ip1dbg(("ip_sioctl_token(%s:%u %p)\n",
11404             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11405         ASSERT(IAM_WRITER_IPIF(ipif));

11407         addrlen = lifr->lifr_addrlen;
11408         /* Only allow for logical unit zero i.e. not on "le0:17" */
11409         if (ipif->ipif_id != 0)
11410                 return (EINVAL);

11412         if (!ipif->ipif_isv6)
11413                 return (EINVAL);
```



```
11415         if (addrlen > IPV6_ABITS)
11416                 return (EINVAL);

11418         v6addr = sin6->sin6_addr;

11420         /*
11421          * The length of the token is the length from the end.  To get
11422          * the proper mask for this, compute the mask of the bits not
11423          * in the token; ie. the prefix, and then xor to get the mask.
11424          */
11425         if (ip_plen_to_mask_v6(IPV6_ABITS - addrlen, &v6mask) == NULL)
11426                 return (EINVAL);
11427         for (i = 0; i < 4; i++) {
11428                 v6mask.s6_addr32[i] ^= (uint32_t)0xffffffff;
11429         }

11431         if (V6_MASK_EQ(v6addr, v6mask, ill->ill_token) &&
11432             ill->ill_token_length == addrlen)
11433                 return (0);     /* No change */

11435         if (ipif->ipif_flags & IPIF_UP) {
11436                 err = ipif_logical_down(ipif, q, mp);
11437                 if (err == EINPROGRESS)
11438                         return (err);
11439                 (void) ipif_down_tail(ipif);
11440                 need_up = B_TRUE;
11441         }
11442         err = ip_sioctl_token_tail(ipif, sin6, addrlen, q, mp, need_up);
11443         return (err);
11444 }

11446 static int
11447 ip_sioctl_token_tail(ipif_t *ipif, sin6_t *sin6, int addrlen, queue_t *q,
11448     mblk_t *mp, boolean_t need_up)
11449 {
11450         in6_addr_t v6addr;
11451         in6_addr_t v6mask;
11452         ill_t   *ill = ipif->ipif_ill;
11453         int     i;
11454         int     err = 0;

11456         ip1dbg(("ip_sioctl_token_tail(%s:%u %p)\n",
11457             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11458         v6addr = sin6->sin6_addr;
11459         /*
11460          * The length of the token is the length from the end.  To get
11461          * the proper mask for this, compute the mask of the bits not
11462          * in the token; ie. the prefix, and then xor to get the mask.
11463          */
11464         (void) ip_plen_to_mask_v6(IPV6_ABITS - addrlen, &v6mask);
11465         for (i = 0; i < 4; i++)
11466                 v6mask.s6_addr32[i] ^= (uint32_t)0xffffffff;

11468         mutex_enter(&ill->ill_lock);
11469         V6_MASK_COPY(v6addr, v6mask, ill->ill_token);
11470         ill->ill_token_length = addrlen;
11471         ill->ill_manual_token = 1;

11473         /* Reconfigure the link-local address based on this new token */
11474         ipif_setlinklocal(ill->ill_ipif);

11476         mutex_exit(&ill->ill_lock);

11478         if (need_up) {
11479                 /*
```

```
11480                  * Now bring the interface back up.  If this
11481                  * is the only IPIF for the ILL, ipif_up
11482                  * will have to re-bind to the device, so
11483                  * we may get back EINPROGRESS, in which
11484                  * case, this IOCTL will get completed in
11485                  * ip_rput_dlpi when we see the DL_BIND_ACK.
11486                  */
11487                 err = ipif_up(ipif, q, mp);
11488                 if (err == EINPROGRESS)
11489                         return (err);
11490         }
11491         return (err);
11492 }

11494 /* ARGSUSED */
11495 int
11496 ip_sioctl_get_token(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11497     ip_ioctl_cmd_t *ipi, void *if_req)
11498 {
11499         ill_t *ill;
11500         sin6_t *sin6 = (sin6_t *)sin;
11501         struct lifreq *lifr = (struct lifreq *)if_req;

11503         ip1dbg(("ip_sioctl_get_token(%s:%u %p)\n",
11504             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11505         if (ipif->ipif_id != 0)
11506                 return (EINVAL);

11508         ill = ipif->ipif_ill;
11509         if (!ill->ill_isv6)
11510                 return (ENXIO);

11512         *sin6 = sin6_null;
11513         sin6->sin6_family = AF_INET6;
11514         ASSERT(!IN6_IS_ADDR_V4MAPPED(&ill->ill_token));
11515         sin6->sin6_addr = ill->ill_token;
11516         lifr->lifr_addrlen = ill->ill_token_length;
11517         return (0);
11518 }

11520 /*
11521  * Set (hardware) link specific information that might override
11522  * what was acquired through the DL_INFO_ACK.
11523  */
11524 /* ARGSUSED */
11525 int
11526 ip_sioctl_lnkinfo(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11527     ip_ioctl_cmd_t *ipi, void *if_req)
11528 {
11529         ill_t           *ill = ipif->ipif_ill;
11530         int             ip_min_mtu;
11531         struct lifreq   *lifr = (struct lifreq *)if_req;
11532         lif_ifinfo_req_t *lir;

11534         ip1dbg(("ip_sioctl_lnkinfo(%s:%u %p)\n",
11535             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11536         lir = &lifr->lifr_ifinfo;
11537         ASSERT(IAM_WRITER_IPIF(ipif));

11539         /* Only allow for logical unit zero i.e. not on "bge0:17" */
11540         if (ipif->ipif_id != 0)
11541                 return (EINVAL);

11543         /* Set interface MTU. */
11544         if (ipif->ipif_isv6)
11545                 ip_min_mtu = IPV6_MIN_MTU;
```



```
11546         else
11547                 ip_min_mtu = IP_MIN_MTU;

11549         /*
11550          * Verify values before we set anything. Allow zero to
11551          * mean unspecified.
11552          *
11553          * XXX We should be able to set the user-defined lir_mtu to some value
11554          * that is greater than ill_current_frag but less than ill_max_frag- the
11555          * ill_max_frag value tells us the max MTU that can be handled by the
11556          * datalink, whereas the ill_current_frag is dynamically computed for
11557          * some link-types like tunnels, based on the tunnel PMTU. However,
11558          * since there is currently no way of distinguishing between
11559          * administratively fixed link mtu values (e.g., those set via
11560          * /sbin/dladm) and dynamically discovered MTUs (e.g., those discovered
11561          * for tunnels) we conservatively choose the  ill_current_frag as the
11562          * upper-bound.
11563          */
11564         if (lir->lir_maxmtu != 0 &&
11565             (lir->lir_maxmtu > ill->ill_current_frag ||
11566             lir->lir_maxmtu < ip_min_mtu))
11567                 return (EINVAL);
11568         if (lir->lir_reachtime != 0 &&
11569             lir->lir_reachtime > ND_MAX_REACHTIME)
11570                 return (EINVAL);
11571         if (lir->lir_reachretrans != 0 &&
11572             lir->lir_reachretrans > ND_MAX_REACHRETRANSTIME)
11573                 return (EINVAL);

11575         mutex_enter(&ill->ill_lock);
11576         /*
11577          * The dce and fragmentation code can handle changes to ill_mtu
11578          * concurrent with sending/fragmenting packets.
11579          */
11580         if (lir->lir_maxmtu != 0)
11581                 ill->ill_user_mtu = lir->lir_maxmtu;

11583         if (lir->lir_reachtime != 0)
11584                 ill->ill_reachable_time = lir->lir_reachtime;

11586         if (lir->lir_reachretrans != 0)
11587                 ill->ill_reachable_retrans_time = lir->lir_reachretrans;

11589         ill->ill_max_hops = lir->lir_maxhops;
11590         ill->ill_max_buf = ND_MAX_Q;
11591         if (!(ill->ill_flags & ILLF_FIXEDMTU) && ill->ill_user_mtu != 0) {
11592                 /*
11593                  * ill_mtu is the actual interface MTU, obtained as the min
11594                  * of user-configured mtu and the value announced by the
11595                  * driver (via DL_NOTE_SDU_SIZE/DL_INFO_ACK). Note that since
11596                  * we have already made the choice of requiring
11597                  * ill_user_mtu < ill_current_frag by the time we get here,
11598                  * the ill_mtu effectively gets assigned to the ill_user_mtu
11599                  * here.
11600                  */
11601                 ill->ill_mtu = MIN(ill->ill_current_frag, ill->ill_user_mtu);
11602                 ill->ill_mc_mtu = MIN(ill->ill_mc_mtu, ill->ill_user_mtu);
11603         }
11604         mutex_exit(&ill->ill_lock);

11606         /*
11607          * Make sure all dce_generation checks find out
11608          * that ill_mtu/ill_mc_mtu has changed.
11609          */
11610         if (!(ill->ill_flags & ILLF_FIXEDMTU) && (lir->lir_maxmtu != 0))
11611                 dce_increment_all_generations(ill->ill_isv6, ill->ill_ipst);
```

```
11613         /*
11614          * Refresh IPMP meta-interface MTU if necessary.
11615          */
11616         if (IS_UNDER_IPMP(ill))
11617                 ipmp_illgrp_refresh_mtu(ill->ill_grp);

11619         return (0);
11620 }

11622 /* ARGSUSED */
11623 int
11624 ip_sioctl_get_lnkinfo(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11625     ip_ioctl_cmd_t *ipi, void *if_req)
11626 {
11627         struct lif_ifinfo_req *lir;
11628         ill_t *ill = ipif->ipif_ill;

11630         ip1dbg(("ip_sioctl_get_lnkinfo(%s:%u %p)\n",
11631             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11632         if (ipif->ipif_id != 0)
11633                 return (EINVAL);

11635         lir = &((struct lifreq *)if_req)->lifr_ifinfo;
11636         lir->lir_maxhops = ill->ill_max_hops;
11637         lir->lir_reachtime = ill->ill_reachable_time;
11638         lir->lir_reachretrans = ill->ill_reachable_retrans_time;
11639         lir->lir_maxmtu = ill->ill_mtu;

11641         return (0);
11642 }

11644 /*
11645  * Return best guess as to the subnet mask for the specified address.
11646  * Based on the subnet masks for all the configured interfaces.
11647  *
11648  * We end up returning a zero mask in the case of default, multicast or
11649  * experimental.
11650  */
11651 static ipaddr_t
11652 ip_subnet_mask(ipaddr_t addr, ipif_t **ipifp, ip_stack_t *ipst)
11653 {
11654         ipaddr_t net_mask;
11655         ill_t   *ill;
11656         ipif_t  *ipif;
11657         ill_walk_context_t ctx;
11658         ipif_t  *fallback_ipif = NULL;

11660         net_mask = ip_net_mask(addr);
11661         if (net_mask == 0) {
11662                 *ipifp = NULL;
11663                 return (0);
11664         }

11666         /* Let's check to see if this is maybe a local subnet route. */
11667         /* this function only applies to IPv4 interfaces */
11668         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
11669         ill = ILL_START_WALK_V4(&ctx, ipst);
11670         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
11671                 mutex_enter(&ill->ill_lock);
11672                 for (ipif = ill->ill_ipif; ipif != NULL;
11673                     ipif = ipif->ipif_next) {
11674                         if (IPIF_IS_CONDEMNED(ipif))
11675                                 continue;
11676                         if (!(ipif->ipif_flags & IPIF_UP))
11677                                 continue;
```



```
11678                         if ((ipif->ipif_subnet & net_mask) ==
11679                             (addr & net_mask)) {
11680                                 /*
11681                                  * Don't trust pt-pt interfaces if there are
11682                                  * other interfaces.
11683                                  */
11684                                 if (ipif->ipif_flags & IPIF_POINTOPOINT) {
11685                                         if (fallback_ipif == NULL) {
11686                                                 ipif_refhold_locked(ipif);
11687                                                 fallback_ipif = ipif;
11688                                         }
11689                                         continue;
11690                                 }

11692                                 /*
11693                                  * Fine. Just assume the same net mask as the
11694                                  * directly attached subnet interface is using.
11695                                  */
11696                                 ipif_refhold_locked(ipif);
11697                                 mutex_exit(&ill->ill_lock);
11698                                 rw_exit(&ipst->ips_ill_g_lock);
11699                                 if (fallback_ipif != NULL)
11700                                         ipif_refrele(fallback_ipif);
11701                                 *ipifp = ipif;
11702                                 return (ipif->ipif_net_mask);
11703                         }
11704                 }
11705                 mutex_exit(&ill->ill_lock);
11706         }
11707         rw_exit(&ipst->ips_ill_g_lock);

11709         *ipifp = fallback_ipif;
11710         return ((fallback_ipif != NULL) ?
11711             fallback_ipif->ipif_net_mask : net_mask);
11712 }

11714 /*
11715  * ip_sioctl_copyin_setup calls ip_wput_ioctl to process the IP_IOCTL ioctl.
11716  */
11717 static void
11718 ip_wput_ioctl(queue_t *q, mblk_t *mp)
11719 {
11720         IOCP    iocp;
11721         ipft_t  *ipft;
11722         ipllc_t *ipllc;
11723         mblk_t  *mp1;
11724         cred_t  *cr;
11725         int     error = 0;
11726         conn_t  *connp;

11728         ip1dbg(("ip_wput_ioctl"));
11729         iocp = (IOCP)mp->b_rptr;
11730         mp1 = mp->b_cont;
11731         if (mp1 == NULL) {
11732                 iocp->ioc_error = EINVAL;
11733                 mp->b_datap->db_type = M_IOCNAK;
11734                 iocp->ioc_count = 0;
11735                 qreply(q, mp);
11736                 return;
11737         }

11739         /*
11740          * These IOCTLs provide various control capabilities to
11741          * upstream agents such as ULPs and processes.  There
11742          * are currently two such IOCTLs implemented.  They
11743          * are used by TCP to provide update information for
```

```
11744          * existing IREs and to forcibly delete an IRE for a
11745          * host that is not responding, thereby forcing an
11746          * attempt at a new route.
11747          */
11748         iocp->ioc_error = EINVAL;
11749         if (!pullupmsg(mp1, sizeof (ipllc->ipllc_cmd)))
11750                 goto done;

11752         ipllc = (ipllc_t *)mp1->b_rptr;
11753         for (ipft = ip_ioctl_ftbl; ipft->ipft_pfi; ipft++) {
11754                 if (ipllc->ipllc_cmd == ipft->ipft_cmd)
11755                         break;
11756         }
11757         /*
11758          * prefer credential from mblk over ioctl;
11759          * see ip_sioctl_copyin_setup
11760          */
11761         cr = msg_getcred(mp, NULL);
11762         if (cr == NULL)
11763                 cr = iocp->ioc_cr;

11765         /*
11766          * Refhold the conn in case the request gets queued up in some lookup
11767          */
11768         ASSERT(CONN_Q(q));
11769         connp = Q_TO_CONN(q);
11770         CONN_INC_REF(connp);
11771         CONN_INC_IOCTLREF(connp);
11772         if (ipft->ipft_pfi &&
11773             ((mp1->b_wptr - mp1->b_rptr) >= ipft->ipft_min_size ||
11774             pullupmsg(mp1, ipft->ipft_min_size))) {
11775                 error = (*ipft->ipft_pfi)(q,
11776                     (ipft->ipft_flags & IPFT_F_SELF_REPLY) ? mp : mp1, cr);
11777         }
11778         if (ipft->ipft_flags & IPFT_F_SELF_REPLY) {
11779                 /*
11780                  * CONN_OPER_PENDING_DONE happens in the function called
11781                  * through ipft_pfi above.
11782                  */
11783                 return;
11784         }

11786         CONN_DEC_IOCTLREF(connp);
11787         CONN_OPER_PENDING_DONE(connp);
11788         if (ipft->ipft_flags & IPFT_F_NO_REPLY) {
11789                 freemsg(mp);
11790                 return;
11791         }
11792         iocp->ioc_error = error;

11794 done:
11795         mp->b_datap->db_type = M_IOCACK;
11796         if (iocp->ioc_error)
11797                 iocp->ioc_count = 0;
11798         qreply(q, mp);
11799 }

11801 /*
11802  * Assign a unique id for the ipif. This is used by sctp_addr.c
11803  * Note: remove if sctp_addr.c is redone to not shadow ill/ipif data structures.
11804  */
11805 static void
11806 ipif_assign_seqid(ipif_t *ipif)
11807 {
11808         ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;
```



```
11810         ipif->ipif_seqid = atomic_add_64_nv(&ipst->ips_ipif_g_seqid, 1);
11811 }

11813 /*
11814  * Clone the contents of 'sipif' to 'dipif'.  Requires that both ipifs are
11815  * administratively down (i.e., no DAD), of the same type, and locked.  Note
11816  * that the clone is complete -- including the seqid -- and the expectation is
11817  * that the caller will either free or overwrite 'sipif' before it's unlocked.
11818  */
11819 static void
11820 ipif_clone(const ipif_t *sipif, ipif_t *dipif)
11821 {
11822         ASSERT(MUTEX_HELD(&sipif->ipif_ill->ill_lock));
11823         ASSERT(MUTEX_HELD(&dipif->ipif_ill->ill_lock));
11824         ASSERT(!(sipif->ipif_flags & (IPIF_UP|IPIF_DUPLICATE)));
11825         ASSERT(!(dipif->ipif_flags & (IPIF_UP|IPIF_DUPLICATE)));
11826         ASSERT(sipif->ipif_ire_type == dipif->ipif_ire_type);

11828         dipif->ipif_flags = sipif->ipif_flags;
11829         dipif->ipif_zoneid = sipif->ipif_zoneid;
11830         dipif->ipif_v6subnet = sipif->ipif_v6subnet;
11831         dipif->ipif_v6lcl_addr = sipif->ipif_v6lcl_addr;
11832         dipif->ipif_v6net_mask = sipif->ipif_v6net_mask;
11833         dipif->ipif_v6brd_addr = sipif->ipif_v6brd_addr;
11834         dipif->ipif_v6pp_dst_addr = sipif->ipif_v6pp_dst_addr;

11836         /*
11837          * As per the comment atop the function, we assume that these sipif
11838          * fields will be changed before sipif is unlocked.
11839          */
11840         dipif->ipif_seqid = sipif->ipif_seqid;
11841         dipif->ipif_state_flags = sipif->ipif_state_flags;
11842 }

11844 /*
11845  * Transfer the contents of 'sipif' to 'dipif', and then free (if 'virgipif'
11846  * is NULL) or overwrite 'sipif' with 'virgipif', which must be a virgin
11847  * (unreferenced) ipif.  Also, if 'sipif' is used by the current xop, then
11848  * transfer the xop to 'dipif'.  Requires that all ipifs are administratively
11849  * down (i.e., no DAD), of the same type, and unlocked.
11850  */
11851 static void
11852 ipif_transfer(ipif_t *sipif, ipif_t *dipif, ipif_t *virgipif)
11853 {
11854         ipsq_t *ipsq = sipif->ipif_ill->ill_phyint->phyint_ipsq;
11855         ipxop_t *ipx = ipsq->ipsq_xop;

11857         ASSERT(sipif != dipif);
11858         ASSERT(sipif != virgipif);

11860         /*
11861          * Grab all of the locks that protect the ipif in a defined order.
11862          */
11863         GRAB_ILL_LOCKS(sipif->ipif_ill, dipif->ipif_ill);

11865         ipif_clone(sipif, dipif);
11866         if (virgipif != NULL) {
11867                 ipif_clone(virgipif, sipif);
11868                 mi_free(virgipif);
11869         }

11871         RELEASE_ILL_LOCKS(sipif->ipif_ill, dipif->ipif_ill);

11873         /*
11874          * Transfer ownership of the current xop, if necessary.
11875          */
```

```
11876         if (ipx->ipx_current_ipif == sipif) {
11877                 ASSERT(ipx->ipx_pending_ipif == NULL);
11878                 mutex_enter(&ipx->ipx_lock);
11879                 ipx->ipx_current_ipif = dipif;
11880                 mutex_exit(&ipx->ipx_lock);
11881         }

11883         if (virgipif == NULL)
11884                 mi_free(sipif);
11885 }

11887 /*
11888  * checks if:
11889  *      - <ill_name>:<ipif_id> is at most LIFNAMSIZ - 1 and
11890  *      - logical interface is within the allowed range
11891  */
11892 static int
11893 is_lifname_valid(ill_t *ill, unsigned int ipif_id)
11894 {
11895         if (snprintf(NULL, 0, "%s:%d", ill->ill_name, ipif_id) >= LIFNAMSIZ)
11896                 return (ENAMETOOLONG);

11898         if (ipif_id >= ill->ill_ipst->ips_ip_addrs_per_if)
11899                 return (ERANGE);
11900         return (0);
11901 }

11903 /*
11904  * Insert the ipif, so that the list of ipifs on the ill will be sorted
11905  * with respect to ipif_id. Note that an ipif with an ipif_id of -1 will
11906  * be inserted into the first space available in the list. The value of
11907  * ipif_id will then be set to the appropriate value for its position.
11908  */
11909 static int
11910 ipif_insert(ipif_t *ipif, boolean_t acquire_g_lock)
11911 {
11912         ill_t *ill;
11913         ipif_t *tipif;
11914         ipif_t **tipifp;
11915         int id, err;
11916         ip_stack_t      *ipst;

11918         ASSERT(ipif->ipif_ill->ill_net_type == IRE_LOOPBACK ||
11919             IAM_WRITER_IPIF(ipif));

11921         ill = ipif->ipif_ill;
11922         ASSERT(ill != NULL);
11923         ipst = ill->ill_ipst;

11925         /*
11926          * In the case of lo0:0 we already hold the ill_g_lock.
11927          * ill_lookup_on_name (acquires ill_g_lock) -> ipif_allocate ->
11928          * ipif_insert.
11929          */
11930         if (acquire_g_lock)
11931                 rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
11932         mutex_enter(&ill->ill_lock);
11933         id = ipif->ipif_id;
11934         tipifp = &(ill->ill_ipif);
11935         if (id == -1) { /* need to find a real id */
11936                 id = 0;
11937                 while ((tipif = *tipifp) != NULL) {
11938                         ASSERT(tipif->ipif_id >= id);
11939                         if (tipif->ipif_id != id)
11940                                 break; /* non-consecutive id */
11941                         id++;
```



```
11942                         tipifp = &(tipif->ipif_next);
11943                 }
11944                 if ((err = is_lifname_valid(ill, id)) != 0) {
11945                         mutex_exit(&ill->ill_lock);
11946                         if (acquire_g_lock)
11947                                 rw_exit(&ipst->ips_ill_g_lock);
11948                         return (err);
11949                 }
11950                 ipif->ipif_id = id; /* assign new id */
11951         } else if ((err = is_lifname_valid(ill, id)) == 0) {
11952                 /* we have a real id; insert ipif in the right place */
11953                 while ((tipif = *tipifp) != NULL) {
11954                         ASSERT(tipif->ipif_id != id);
11955                         if (tipif->ipif_id > id)
11956                                 break; /* found correct location */
11957                         tipifp = &(tipif->ipif_next);
11958                 }
11959         } else {
11960                 mutex_exit(&ill->ill_lock);
11961                 if (acquire_g_lock)
11962                         rw_exit(&ipst->ips_ill_g_lock);
11963                 return (err);
11964         }

11966         ASSERT(tipifp != &(ill->ill_ipif) || id == 0);

11968         ipif->ipif_next = tipif;
11969         *tipifp = ipif;
11970         mutex_exit(&ill->ill_lock);
11971         if (acquire_g_lock)
11972                 rw_exit(&ipst->ips_ill_g_lock);

11974         return (0);
11975 }

11977 static void
11978 ipif_remove(ipif_t *ipif)
11979 {
11980         ipif_t  **ipifp;
11981         ill_t   *ill = ipif->ipif_ill;

11983         ASSERT(RW_WRITE_HELD(&ill->ill_ipst->ips_ill_g_lock));

11985         mutex_enter(&ill->ill_lock);
11986         ipifp = &ill->ill_ipif;
11987         for (; *ipifp != NULL; ipifp = &ipifp[0]->ipif_next) {
11988                 if (*ipifp == ipif) {
11989                         *ipifp = ipif->ipif_next;
11990                         break;
11991                 }
11992         }
11993         mutex_exit(&ill->ill_lock);
11994 }

11996 /*
11997  * Allocate and initialize a new interface control structure.  (Always
11998  * called as writer.)
11999  * When ipif_allocate() is called from ip_ll_subnet_defaults, the ill
12000  * is not part of the global linked list of ills. ipif_seqid is unique
12001  * in the system and to preserve the uniqueness, it is assigned only
12002  * when ill becomes part of the global list. At that point ill will
12003  * have a name. If it doesn't get assigned here, it will get assigned
12004  * in ipif_set_values() as part of SIOCSLIFNAME processing.
12005  * Aditionally, if we come here from ip_ll_subnet_defaults, we don't set
12006  * the interface flags or any other information from the DL_INFO_ACK for
12007  * DL_STYLE2 drivers (initialize == B_FALSE), since we won't have them at
```

```
 * this point. The flags etc. will be set in ip_ll_subnet_defaults when the
 * second DL_INFO_ACK comes in from the driver.
 */
static ipif_t *
ipif_allocate(ill_t *ill, int id, uint_t ire_type, boolean_t initialize,
    boolean_t insert, int *errorp)
{
        int err;
        ipif_t  *ipif;
        ip_stack_t *ipst = ill->ill_ipst;

        ip1dbg(("ipif_allocate(%s:%d ill %p)\n",
            ill->ill_name, id, (void *)ill));
        ASSERT(ire_type == IRE_LOOPBACK || IAM_WRITER_ILL(ill));

        if (errorp != NULL)
                *errorp = 0;

        if ((ipif = mi_alloc(sizeof (ipif_t), BPRI_MED)) == NULL) {
                if (errorp != NULL)
                        *errorp = ENOMEM;
                return (NULL);
        }
        *ipif = ipif_zero;      /* start clean */

        ipif->ipif_ill = ill;
        ipif->ipif_id = id;     /* could be -1 */
        /*
         * Inherit the zoneid from the ill; for the shared stack instance
         * this is always the global zone
         */
        ipif->ipif_zoneid = ill->ill_zoneid;

        ipif->ipif_refcnt = 0;

        if (insert) {
                if ((err = ipif_insert(ipif, ire_type != IRE_LOOPBACK)) != 0) {
                        mi_free(ipif);
                        if (errorp != NULL)
                                *errorp = err;
                        return (NULL);
                }
                /* -1 id should have been replaced by real id */
                id = ipif->ipif_id;
                ASSERT(id >= 0);
        }

        if (ill->ill_name[0] != '\0')
                ipif_assign_seqid(ipif);

        /*
         * If this is the zeroth ipif on the IPMP ill, create the illgrp
         * (which must not exist yet because the zeroth ipif is created once
         * per ill).  However, do not not link it to the ipmp_grp_t until
         * I_PLINK is called; see ip_sioctl_plink_ipmp() for details.
         */
        if (id == 0 && IS_IPMP(ill)) {
                if (ipmp_illgrp_create(ill) == NULL) {
                        if (insert) {
                                rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
                                ipif_remove(ipif);
                                rw_exit(&ipst->ips_ill_g_lock);
                        }
                        mi_free(ipif);
                        if (errorp != NULL)
                                *errorp = ENOMEM;
```



```
                        return (NULL);
                }
        }

        /*
         * We grab ill_lock to protect the flag changes.  The ipif is still
         * not up and can't be looked up until the ioctl completes and the
         * IPIF_CHANGING flag is cleared.
         */
        mutex_enter(&ill->ill_lock);

        ipif->ipif_ire_type = ire_type;

        if (ipif->ipif_isv6) {
                ill->ill_flags |= ILLF_IPV6;
        } else {
                ipaddr_t inaddr_any = INADDR_ANY;

                ill->ill_flags |= ILLF_IPV4;

                /* Keep the IN6_IS_ADDR_V4MAPPED assertions happy */
                IN6_IPADDR_TO_V4MAPPED(inaddr_any,
                    &ipif->ipif_v6lcl_addr);
                IN6_IPADDR_TO_V4MAPPED(inaddr_any,
                    &ipif->ipif_v6subnet);
                IN6_IPADDR_TO_V4MAPPED(inaddr_any,
                    &ipif->ipif_v6net_mask);
                IN6_IPADDR_TO_V4MAPPED(inaddr_any,
                    &ipif->ipif_v6brd_addr);
                IN6_IPADDR_TO_V4MAPPED(inaddr_any,
                    &ipif->ipif_v6pp_dst_addr);
        }

        /*
         * Don't set the interface flags etc. now, will do it in
         * ip_ll_subnet_defaults.
         */
        if (!initialize)
                goto out;

        /*
         * NOTE: The IPMP meta-interface is special-cased because it starts
         * with no underlying interfaces (and thus an unknown broadcast
         * address length), but all interfaces that can be placed into an IPMP
         * group are required to be broadcast-capable.
         */
        if (ill->ill_bcast_addr_length != 0 || IS_IPMP(ill)) {
                /*
                 * Later detect lack of DLPI driver multicast capability by
                 * catching DL_ENABMULTI_REQ errors in ip_rput_dlpi().
                 */
                ill->ill_flags |= ILLF_MULTICAST;
                if (!ipif->ipif_isv6)
                        ipif->ipif_flags |= IPIF_BROADCAST;
        } else {
                if (ill->ill_net_type != IRE_LOOPBACK) {
                        if (ipif->ipif_isv6)
                                /*
                                 * Note: xresolv interfaces will eventually need
                                 * NOARP set here as well, but that will require
                                 * those external resolvers to have some
                                 * knowledge of that flag and act appropriately.
                                 * Not to be changed at present.
                                 */
                                ill->ill_flags |= ILLF_NONUD;
                        else
```

```
12140                                   ill->ill_flags |= ILLF_NOARP;
12141                           }
12142                           if (ill->ill_phys_addr_length == 0) {
12143                                   if (IS_VNI(ill)) {
12144                                           ipif->ipif_flags |= IPIF_NOXMIT;
12145                                   } else {
12146                                           /* pt-pt supports multicast. */
12147                                           ill->ill_flags |= ILLF_MULTICAST;
12148                                           if (ill->ill_net_type != IRE_LOOPBACK)
12149                                                   ipif->ipif_flags |= IPIF_POINTOPOINT;
12150                                   }
12151                           }
12152                   }
12153           out:
12154                   mutex_exit(&ill->ill_lock);
12155                   return (ipif);
12156           }

12158           /*
12159            * Remove the neighbor cache entries associated with this logical
12160            * interface.
12161            */
12162           int
12163           ipif_arp_down(ipif_t *ipif)
12164           {
12165                   ill_t   *ill = ipif->ipif_ill;
12166                   int     err = 0;

12168                   ip1dbg(("ipif_arp_down(%s:%u)\n", ill->ill_name, ipif->ipif_id));
12169                   ASSERT(IAM_WRITER_IPIF(ipif));

12171                   DTRACE_PROBE3(ipif__downup, char *, "ipif_arp_down",
12172                       ill_t *, ill, ipif_t *, ipif);
12173                   ipif_nce_down(ipif);

12175                   /*
12176                    * If this is the last ipif that is going down and there are no
12177                    * duplicate addresses we may yet attempt to re-probe, then we need to
12178                    * clean up ARP completely.
12179                    */
12180                   if (ill->ill_ipif_up_count == 0 && ill->ill_ipif_dup_count == 0 &&
12181                       !ill->ill_logical_down && ill->ill_net_type == IRE_IF_RESOLVER) {
12182                           /*
12183                            * If this was the last ipif on an IPMP interface, purge any
12184                            * static ARP entries associated with it.
12185                            */
12186                           if (IS_IPMP(ill))
12187                                   ipmp_illgrp_refresh_arpent(ill->ill_grp);

12189                           /* UNBIND, DETACH */
12190                           err = arp_ll_down(ill);
12191                   }

12193                   return (err);
12194           }

12196           /*
12197            * Get the resolver set up for a new IP address.  (Always called as writer.)
12198            * Called both for IPv4 and IPv6 interfaces, though it only does some
12199            * basic DAD related initialization for IPv6. Honors ILLF_NOARP.
12200            *
12201            * The enumerated value res_act tunes the behavior:
12202            *      * Res_act_initial: set up all the resolver structures for a new
12203            *        IP address.
12204            *      * Res_act_defend: tell ARP that it needs to send a single gratuitous
12205            *        ARP message in defense of the address.
```



```
12206            *      * Res_act_rebind: tell ARP to change the hardware address for an IP
12207            *        address (and issue gratuitous ARPs).  Used by ipmp_ill_bind_ipif().
12208            *
12209            * Returns zero on success, or an errno upon failure.
12210            */
12211           int
12212           ipif_resolver_up(ipif_t *ipif, enum ip_resolver_action res_act)
12213           {
12214                   ill_t           *ill = ipif->ipif_ill;
12215                   int             err;
12216                   boolean_t       was_dup;

12218                   ip1dbg(("ipif_resolver_up(%s:%u) flags 0x%x\n",
12219                       ill->ill_name, ipif->ipif_id, (uint_t)ipif->ipif_flags));
12220                   ASSERT(IAM_WRITER_IPIF(ipif));

12222                   was_dup = B_FALSE;
12223                   if (res_act == Res_act_initial) {
12224                           ipif->ipif_addr_ready = 0;
12225                           /*
12226                            * We're bringing an interface up here.  There's no way that we
12227                            * should need to shut down ARP now.
12228                            */
12229                           mutex_enter(&ill->ill_lock);
12230                           if (ipif->ipif_flags & IPIF_DUPLICATE) {
12231                                   ipif->ipif_flags &= ~IPIF_DUPLICATE;
12232                                   ill->ill_ipif_dup_count--;
12233                                   was_dup = B_TRUE;
12234                           }
12235                           mutex_exit(&ill->ill_lock);
12236                   }
12237                   if (ipif->ipif_recovery_id != 0)
12238                           (void) untimeout(ipif->ipif_recovery_id);
12239                   ipif->ipif_recovery_id = 0;
12240                   if (ill->ill_net_type != IRE_IF_RESOLVER) {
12241                           ipif->ipif_addr_ready = 1;
12242                           return (0);
12243                   }
12244                   /* NDP will set the ipif_addr_ready flag when it's ready */
12245                   if (ill->ill_isv6)
12246                           return (0);

12248                   err = ipif_arp_up(ipif, res_act, was_dup);
12249                   return (err);
12250           }

12252           /*
12253            * This routine restarts IPv4/IPv6 duplicate address detection (DAD)
12254            * when a link has just gone back up.
12255            */
12256           static void
12257           ipif_nce_start_dad(ipif_t *ipif)
12258           {
12259                   ncec_t *ncec;
12260                   ill_t *ill = ipif->ipif_ill;
12261                   boolean_t isv6 = ill->ill_isv6;

12263                   if (isv6) {
12264                           ncec = ncec_lookup_illgrp_v6(ipif->ipif_ill,
12265                               &ipif->ipif_v6lcl_addr);
12266                   } else {
12267                           ipaddr_t v4addr;

12269                           if (ill->ill_net_type != IRE_IF_RESOLVER ||
12270                               (ipif->ipif_flags & IPIF_UNNUMBERED) ||
12271                               ipif->ipif_lcl_addr == INADDR_ANY) {
```

```
12272                         /*
12273                          * If we can't contact ARP for some reason,
12274                          * that's not really a problem.  Just send
12275                          * out the routing socket notification that
12276                          * DAD completion would have done, and continue.
12277                          */
12278                         ipif_mask_reply(ipif);
12279                         ipif_up_notify(ipif);
12280                         ipif->ipif_addr_ready = 1;
12281                         return;
12282                 }

12284                 IN6_V4MAPPED_TO_IPADDR(&ipif->ipif_v6lcl_addr, v4addr);
12285                 ncec = ncec_lookup_illgrp_v4(ipif->ipif_ill, &v4addr);
12286         }

12288         if (ncec == NULL) {
12289                 ip1dbg(("couldn't find ncec for ipif %p leaving !ready\n",
12290                     (void *)ipif));
12291                 return;
12292         }
12293         if (!nce_restart_dad(ncec)) {
12294                 /*
12295                  * If we can't restart DAD for some reason, that's not really a
12296                  * problem.  Just send out the routing socket notification that
12297                  * DAD completion would have done, and continue.
12298                  */
12299                 ipif_up_notify(ipif);
12300                 ipif->ipif_addr_ready = 1;
12301         }
12302         ncec_refrele(ncec);
12303 }

12305 /*
12306  * Restart duplicate address detection on all interfaces on the given ill.
12307  *
12308  * This is called when an interface transitions from down to up
12309  * (DL_NOTE_LINK_UP) or up to down (DL_NOTE_LINK_DOWN).
12310  *
12311  * Note that since the underlying physical link has transitioned, we must cause
12312  * at least one routing socket message to be sent here, either via DAD
12313  * completion or just by default on the first ipif.  (If we don't do this, then
12314  * in.mpathd will see long delays when doing link-based failure recovery.)
12315  */
12316 void
12317 ill_restart_dad(ill_t *ill, boolean_t went_up)
12318 {
12319         ipif_t *ipif;

12321         if (ill == NULL)
12322                 return;

12324         /*
12325          * If layer two doesn't support duplicate address detection, then just
12326          * send the routing socket message now and be done with it.
12327          */
12328         if (!ill->ill_isv6 && arp_no_defense) {
12329                 ip_rts_ifmsg(ill->ill_ipif, RTSQ_DEFAULT);
12330                 return;
12331         }

12333         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
12334                 if (went_up) {

12336                         if (ipif->ipif_flags & IPIF_UP) {
12337                                 ipif_nce_start_dad(ipif);
```



```
12338                         } else if (ipif->ipif_flags & IPIF_DUPLICATE) {
12339                                 /*
12340                                  * kick off the bring-up process now.
12341                                  */
12342                                 ipif_do_recovery(ipif);
12343                         } else {
12344                                 /*
12345                                  * Unfortunately, the first ipif is "special"
12346                                  * and represents the underlying ill in the
12347                                  * routing socket messages.  Thus, when this
12348                                  * one ipif is down, we must still notify so
12349                                  * that the user knows the IFF_RUNNING status
12350                                  * change.  (If the first ipif is up, then
12351                                  * we'll handle eventual routing socket
12352                                  * notification via DAD completion.)
12353                                  */
12354                                 if (ipif == ill->ill_ipif) {
12355                                         ip_rts_ifmsg(ill->ill_ipif,
12356                                             RTSQ_DEFAULT);
12357                                 }
12358                         }
12359                 } else {
12360                         /*
12361                          * After link down, we'll need to send a new routing
12362                          * message when the link comes back, so clear
12363                          * ipif_addr_ready.
12364                          */
12365                         ipif->ipif_addr_ready = 0;
12366                 }
12367         }

12369         /*
12370          * If we've torn down links, then notify the user right away.
12371          */
12372         if (!went_up)
12373                 ip_rts_ifmsg(ill->ill_ipif, RTSQ_DEFAULT);
12374 }

12376 static void
12377 ipsq_delete(ipsq_t *ipsq)
12378 {
12379         ipxop_t *ipx = ipsq->ipsq_xop;

12381         ipsq->ipsq_ipst = NULL;
12382         ASSERT(ipsq->ipsq_phyint == NULL);
12383         ASSERT(ipsq->ipsq_xop != NULL);
12384         ASSERT(ipsq->ipsq_xopq_mphead == NULL && ipx->ipx_mphead == NULL);
12385         ASSERT(ipx->ipx_pending_mp == NULL);
12386         kmem_free(ipsq, sizeof (ipsq_t));
12387 }

12389 static int
12390 ill_up_ipifs_on_ill(ill_t *ill, queue_t *q, mblk_t *mp)
12391 {
12392         int err = 0;
12393         ipif_t *ipif;

12395         if (ill == NULL)
12396                 return (0);

12398         ASSERT(IAM_WRITER_ILL(ill));
12399         ill->ill_up_ipifs = B_TRUE;
12400         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
12401                 if (ipif->ipif_was_up) {
12402                         if (!(ipif->ipif_flags & IPIF_UP))
12403                                 err = ipif_up(ipif, q, mp);
```

```
12404                         ipif->ipif_was_up = B_FALSE;
12405                         if (err != 0) {
12406                                 ASSERT(err == EINPROGRESS);
12407                                 return (err);
12408                         }
12409                 }
12410         }
12411         ill->ill_up_ipifs = B_FALSE;
12412         return (0);
12413 }

12415 /*
12416  * This function is called to bring up all the ipifs that were up before
12417  * bringing the ill down via ill_down_ipifs().
12418  */
12419 int
12420 ill_up_ipifs(ill_t *ill, queue_t *q, mblk_t *mp)
12421 {
12422         int err;

12424         ASSERT(IAM_WRITER_ILL(ill));

12426         if (ill->ill_replumbing) {
12427                 ill->ill_replumbing = 0;
12428                 /*
12429                  * Send down REPLUMB_DONE notification followed by the
12430                  * BIND_REQ on the arp stream.
12431                  */
12432                 if (!ill->ill_isv6)
12433                         arp_send_replumb_conf(ill);
12434         }
12435         err = ill_up_ipifs_on_ill(ill->ill_phyint->phyint_illv4, q, mp);
12436         if (err != 0)
12437                 return (err);

12439         return (ill_up_ipifs_on_ill(ill->ill_phyint->phyint_illv6, q, mp));
12440 }

12442 /*
12443  * Bring down any IPIF_UP ipifs on ill. If "logical" is B_TRUE, we bring
12444  * down the ipifs without sending DL_UNBIND_REQ to the driver.
12445  */
12446 static void
12447 ill_down_ipifs(ill_t *ill, boolean_t logical)
12448 {
12449         ipif_t *ipif;

12451         ASSERT(IAM_WRITER_ILL(ill));

12453         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
12454                 /*
12455                  * We go through the ipif_down logic even if the ipif
12456                  * is already down, since routes can be added based
12457                  * on down ipifs. Going through ipif_down once again
12458                  * will delete any IREs created based on these routes.
12459                  */
12460                 if (ipif->ipif_flags & IPIF_UP)
12461                         ipif->ipif_was_up = B_TRUE;

12463                 if (logical) {
12464                         (void) ipif_logical_down(ipif, NULL, NULL);
12465                         ipif_non_duplicate(ipif);
12466                         (void) ipif_down_tail(ipif);
12467                 } else {
12468                         (void) ipif_down(ipif, NULL, NULL);
12469                 }
```



```
12470         }
12471 }

12473 /*
12474  * Redo source address selection.  This makes IXAF_VERIFY_SOURCE take
12475  * a look again at valid source addresses.
12476  * This should be called each time after the set of source addresses has been
12477  * changed.
12478  */
12479 void
12480 ip_update_source_selection(ip_stack_t *ipst)
12481 {
12482         /* We skip past SRC_GENERATION_VERIFY */
12483         if (atomic_add_32_nv(&ipst->ips_src_generation, 1) ==
12484             SRC_GENERATION_VERIFY)
12485                 atomic_add_32(&ipst->ips_src_generation, 1);
12486 }

12488 /*
12489  * Finish the group join started in ip_sioctl_groupname().
12490  */
12491 /* ARGSUSED */
12492 static void
12493 ip_join_illgrps(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy)
12494 {
12495         ill_t           *ill = q->q_ptr;
12496         phyint_t        *phyi = ill->ill_phyint;
12497         ipmp_grp_t      *grp = phyi->phyint_grp;
12498         ip_stack_t      *ipst = ill->ill_ipst;

12500         /* IS_UNDER_IPMP() won't work until ipmp_ill_join_illgrp() is called */
12501         ASSERT(!IS_IPMP(ill) && grp != NULL);
12502         ASSERT(IAM_WRITER_IPSQ(ipsq));

12504         if (phyi->phyint_illv4 != NULL) {
12505                 rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);
12506                 VERIFY(grp->gr_pendv4-- > 0);
12507                 rw_exit(&ipst->ips_ipmp_lock);
12508                 ipmp_ill_join_illgrp(phyi->phyint_illv4, grp->gr_v4);
12509         }
12510         if (phyi->phyint_illv6 != NULL) {
12511                 rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);
12512                 VERIFY(grp->gr_pendv6-- > 0);
12513                 rw_exit(&ipst->ips_ipmp_lock);
12514                 ipmp_ill_join_illgrp(phyi->phyint_illv6, grp->gr_v6);
12515         }
12516         freemsg(mp);
12517 }

12519 /*
12520  * Process an SIOCSLIFGROUPNAME request.
12521  */
12522 /* ARGSUSED */
12523 int
12524 ip_sioctl_groupname(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
12525     ip_ioctl_cmd_t *ipip, void *ifreq)
12526 {
12527         struct lifreq   *lifr = ifreq;
12528         ill_t           *ill = ipif->ipif_ill;
12529         ip_stack_t      *ipst = ill->ill_ipst;
12530         phyint_t        *phyi = ill->ill_phyint;
12531         ipmp_grp_t      *grp = phyi->phyint_grp;
12532         mblk_t          *ipsq_mp;
12533         int             err = 0;

12535         /*
```

```
12536          * Note that phyint_grp can only change here, where we're exclusive.
12537          */
12538         ASSERT(IAM_WRITER_ILL(ill));

12540         if (ipif->ipif_id != 0 || ill->ill_usesrc_grp_next != NULL ||
12541             (phyi->phyint_flags & PHYI_VIRTUAL))
12542                 return (EINVAL);

12544         lifr->lifr_groupname[LIFGRNAMSIZ - 1] = '\0';

12546         rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);

12548         /*
12549          * If the name hasn't changed, there's nothing to do.
12550          */
12551         if (grp != NULL && strcmp(grp->gr_name, lifr->lifr_groupname) == 0)
12552                 goto unlock;

12554         /*
12555          * Handle requests to rename an IPMP meta-interface.
12556          *
12557          * Note that creation of the IPMP meta-interface is handled in
12558          * userland through the standard plumbing sequence.  As part of the
12559          * plumbing the IPMP meta-interface, its initial groupname is set to
12560          * the name of the interface (see ipif_set_values_tail()).
12561          */
12562         if (IS_IPMP(ill)) {
12563                 err = ipmp_grp_rename(grp, lifr->lifr_groupname);
12564                 goto unlock;
12565         }

12567         /*
12568          * Handle requests to add or remove an IP interface from a group.
12569          */
12570         if (lifr->lifr_groupname[0] != '\0') {                  /* add */
12571                 /*
12572                  * Moves are handled by first removing the interface from
12573                  * its existing group, and then adding it to another group.
12574                  * So, fail if it's already in a group.
12575                  */
12576                 if (IS_UNDER_IPMP(ill)) {
12577                         err = EALREADY;
12578                         goto unlock;
12579                 }

12581                 grp = ipmp_grp_lookup(lifr->lifr_groupname, ipst);
12582                 if (grp == NULL) {
12583                         err = ENOENT;
12584                         goto unlock;
12585                 }

12587                 /*
12588                  * Check if the phyint and its ills are suitable for
12589                  * inclusion into the group.
12590                  */
12591                 if ((err = ipmp_grp_vet_phyint(grp, phyi)) != 0)
12592                         goto unlock;

12594                 /*
12595                  * Checks pass; join the group, and enqueue the remaining
12596                  * illgrp joins for when we've become part of the group xop
12597                  * and are exclusive across its IPSQs.  Since qwriter_ip()
12598                  * requires an mblk_t to scribble on, and since 'mp' will be
12599                  * freed as part of completing the ioctl, allocate another.
12600                  */
12601                 if ((ipsq_mp = allocb(0, BPRI_MED)) == NULL) {
```



```
12602                         err = ENOMEM;
12603                         goto unlock;
12604                 }

12606                 /*
12607                  * Before we drop ipmp_lock, bump gr_pend* to ensure that the
12608                  * IPMP meta-interface ills needed by 'phyi' cannot go away
12609                  * before ip_join_illgrps() is called back.  See the comments
12610                  * in ip_sioctl_plink_ipmp() for more.
12611                  */
12612                 if (phyi->phyint_illv4 != NULL)
12613                         grp->gr_pendv4++;
12614                 if (phyi->phyint_illv6 != NULL)
12615                         grp->gr_pendv6++;

12617                 rw_exit(&ipst->ips_ipmp_lock);

12619                 ipmp_phyint_join_grp(phyi, grp);
12620                 ill_refhold(ill);
12621                 qwriter_ip(ill, ill->ill_rq, ipsq_mp, ip_join_illgrps,
12622                     SWITCH_OP, B_FALSE);
12623                 return (0);
12624         } else {
12625                 /*
12626                  * Request to remove the interface from a group.  If the
12627                  * interface is not in a group, this trivially succeeds.
12628                  */
12629                 rw_exit(&ipst->ips_ipmp_lock);
12630                 if (IS_UNDER_IPMP(ill))
12631                         ipmp_phyint_leave_grp(phyi);
12632                 return (0);
12633         }
12634 unlock:
12635         rw_exit(&ipst->ips_ipmp_lock);
12636         return (err);
12637 }

12639 /*
12640  * Process an SIOCGLIFBINDING request.
12641  */
12642 /* ARGSUSED */
12643 int
12644 ip_sioctl_get_binding(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
12645     ip_ioctl_cmd_t *ipip, void *ifreq)
12646 {
12647         ill_t           *ill;
12648         struct lifreq   *lifr = ifreq;
12649         ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;

12651         if (!IS_IPMP(ipif->ipif_ill))
12652                 return (EINVAL);

12654         rw_enter(&ipst->ips_ipmp_lock, RW_READER);
12655         if ((ill = ipif->ipif_bound_ill) == NULL)
12656                 lifr->lifr_binding[0] = '\0';
12657         else
12658                 (void) strlcpy(lifr->lifr_binding, ill->ill_name, LIFNAMSIZ);
12659         rw_exit(&ipst->ips_ipmp_lock);
12660         return (0);
12661 }

12663 /*
12664  * Process an SIOCGLIFGROUPNAME request.
12665  */
12666 /* ARGSUSED */
12667 int
```

```
12668 ip_sioctl_get_groupname(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
12669     ip_ioctl_cmd_t *ipip, void *ifreq)
12670 {
12671         ipmp_grp_t      *grp;
12672         struct lifreq   *lifr = ifreq;
12673         ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;

12675         rw_enter(&ipst->ips_ipmp_lock, RW_READER);
12676         if ((grp = ipif->ipif_ill->ill_phyint->phyint_grp) == NULL)
12677                 lifr->lifr_groupname[0] = '\0';
12678         else
12679                 (void) strlcpy(lifr->lifr_groupname, grp->gr_name, LIFGRNAMSIZ);
12680         rw_exit(&ipst->ips_ipmp_lock);
12681         return (0);
12682 }

12684 /*
12685  * Process an SIOCGLIFGROUPINFO request.
12686  */
12687 /* ARGSUSED */
12688 int
12689 ip_sioctl_groupinfo(ipif_t *dummy_ipif, sin_t *sin, queue_t *q, mblk_t *mp,
12690     ip_ioctl_cmd_t *ipip, void *dummy)
12691 {
12692         ipmp_grp_t      *grp;
12693         lifgroupinfo_t  *lifgr;
12694         ip_stack_t      *ipst = CONNQ_TO_IPST(q);

12696         /* ip_wput_nondata() verified mp->b_cont->b_cont */
12697         lifgr = (lifgroupinfo_t *)mp->b_cont->b_cont->b_rptr;
12698         lifgr->gi_grname[LIFGRNAMSIZ - 1] = '\0';

12700         rw_enter(&ipst->ips_ipmp_lock, RW_READER);
12701         if ((grp = ipmp_grp_lookup(lifgr->gi_grname, ipst)) == NULL) {
12702                 rw_exit(&ipst->ips_ipmp_lock);
12703                 return (ENOENT);
12704         }
12705         ipmp_grp_info(grp, lifgr);
12706         rw_exit(&ipst->ips_ipmp_lock);
12707         return (0);
12708 }

12710 static void
12711 ill_dl_down(ill_t *ill)
12712 {
12713         DTRACE_PROBE2(ill__downup, char *, "ill_dl_down", ill_t *, ill);

12715         /*
12716          * The ill is down; unbind but stay attached since we're still
12717          * associated with a PPA. If we have negotiated DLPI capabilites
12718          * with the data link service provider (IDS_OK) then reset them.
12719          * The interval between unbinding and rebinding is potentially
12720          * unbounded hence we cannot assume things will be the same.
12721          * The DLPI capabilities will be probed again when the data link
12722          * is brought up.
12723          */
12724         mblk_t  *mp = ill->ill_unbind_mp;

12726         ip1dbg(("ill_dl_down(%s)\n", ill->ill_name));

12728         if (!ill->ill_replumbing) {
12729                 /* Free all ilms for this ill */
12730                 update_conn_ill(ill, ill->ill_ipst);
12731         } else {
12732                 ill_leave_multicast(ill);
12733         }
```



```
12735         ill->ill_unbind_mp = NULL;
12736         if (mp != NULL) {
12737                 ip1dbg(("ill_dl_down: %s (%u) for %s\n",
12738                     dl_primstr(*(int *)mp->b_rptr), *(int *)mp->b_rptr,
12739                     ill->ill_name));
12740                 mutex_enter(&ill->ill_lock);
12741                 ill->ill_state_flags |= ILL_DL_UNBIND_IN_PROGRESS;
12742                 mutex_exit(&ill->ill_lock);
12743                 /*
12744                  * ip_rput does not pass up normal (M_PROTO) DLPI messages
12745                  * after ILL_CONDEMNED is set. So in the unplumb case, we call
12746                  * ill_capability_dld_disable disable rightaway. If this is not
12747                  * an unplumb operation then the disable happens on receipt of
12748                  * the capab ack via ip_rput_dlpi_writer ->
12749                  * ill_capability_ack_thr. In both cases the order of
12750                  * the operations seen by DLD is capability disable followed
12751                  * by DL_UNBIND. Also the DLD capability disable needs a
12752                  * cv_wait'able context.
12753                  */
12754                 if (ill->ill_state_flags & ILL_CONDEMNED)
12755                         ill_capability_dld_disable(ill);
12756                 ill_capability_reset(ill, B_FALSE);
12757                 ill_dlpi_send(ill, mp);
12758         }
12759         mutex_enter(&ill->ill_lock);
12760         ill->ill_dl_up = 0;
12761         ill_nic_event_dispatch(ill, 0, NE_DOWN, NULL, 0);
12762         mutex_exit(&ill->ill_lock);
12763 }

12765 void
12766 ill_dlpi_dispatch(ill_t *ill, mblk_t *mp)
12767 {
12768         union DL_primitives *dlp;
12769         t_uscalar_t prim;
12770         boolean_t waitack = B_FALSE;

12772         ASSERT(DB_TYPE(mp) == M_PROTO || DB_TYPE(mp) == M_PCPROTO);

12774         dlp = (union DL_primitives *)mp->b_rptr;
12775         prim = dlp->dl_primitive;

12777         ip1dbg(("ill_dlpi_dispatch: sending %s (%u) to %s\n",
12778             dl_primstr(prim), prim, ill->ill_name));

12780         switch (prim) {
12781         case DL_PHYS_ADDR_REQ:
12782         {
12783                 dl_phys_addr_req_t *dlpap = (dl_phys_addr_req_t *)mp->b_rptr;
12784                 ill->ill_phys_addr_pend = dlpap->dl_addr_type;
12785                 break;
12786         }
12787         case DL_BIND_REQ:
12788                 mutex_enter(&ill->ill_lock);
12789                 ill->ill_state_flags &= ~ILL_DL_UNBIND_IN_PROGRESS;
12790                 mutex_exit(&ill->ill_lock);
12791                 break;
12792         }

12794         /*
12795          * Except for the ACKs for the M_PCPROTO messages, all other ACKs
12796          * are dropped by ip_rput() if ILL_CONDEMNED is set. Therefore
12797          * we only wait for the ACK of the DL_UNBIND_REQ.
12798          */
12799         mutex_enter(&ill->ill_lock);
```

```
12800         if (!(ill->ill_state_flags & ILL_CONDEMNED) ||
12801             (prim == DL_UNBIND_REQ)) {
12802                 ill->ill_dlpi_pending = prim;
12803                 waitack = B_TRUE;
12804         }

12806         mutex_exit(&ill->ill_lock);
12807         DTRACE_PROBE3(ill__dlpi, char *, "ill_dlpi_dispatch",
12808             char *, dl_primstr(prim), ill_t *, ill);
12809         putnext(ill->ill_wq, mp);

12811         /*
12812          * There is no ack for DL_NOTIFY_CONF messages
12813          */
12814         if (waitack && prim == DL_NOTIFY_CONF)
12815                 ill_dlpi_done(ill, prim);
12816 }

12818 /*
12819  * Helper function for ill_dlpi_send().
12820  */
12821 /* ARGSUSED */
12822 static void
12823 ill_dlpi_send_writer(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *arg)
12824 {
12825         ill_dlpi_send(q->q_ptr, mp);
12826 }

12828 /*
12829  * Send a DLPI control message to the driver but make sure there
12830  * is only one outstanding message. Uses ill_dlpi_pending to tell
12831  * when it must queue. ip_rput_dlpi_writer calls ill_dlpi_done()
12832  * when an ACK or a NAK is received to process the next queued message.
12833  */
12834 void
12835 ill_dlpi_send(ill_t *ill, mblk_t *mp)
12836 {
12837         mblk_t **mpp;

12839         ASSERT(DB_TYPE(mp) == M_PROTO || DB_TYPE(mp) == M_PCPROTO);

12841         /*
12842          * To ensure that any DLPI requests for current exclusive operation
12843          * are always completely sent before any DLPI messages for other
12844          * operations, require writer access before enqueuing.
12845          */
12846         if (!IAM_WRITER_ILL(ill)) {
12847                 ill_refhold(ill);
12848                 /* qwriter_ip() does the ill_refrele() */
12849                 qwriter_ip(ill, ill->ill_wq, mp, ill_dlpi_send_writer,
12850                     NEW_OP, B_TRUE);
12851                 return;
12852         }

12854         mutex_enter(&ill->ill_lock);
12855         if (ill->ill_dlpi_pending != DL_PRIM_INVAL) {
12856                 /* Must queue message. Tail insertion */
12857                 mpp = &ill->ill_dlpi_deferred;
12858                 while (*mpp != NULL)
12859                         mpp = &((*mpp)->b_next);

12861                 ip1dbg(("ill_dlpi_send: deferring request for %s "
12862                     "while %s pending\n", ill->ill_name,
12863                     dl_primstr(ill->ill_dlpi_pending)));

12865                 *mpp = mp;
```



```
12866                 mutex_exit(&ill->ill_lock);
12867                 return;
12868         }
12869         mutex_exit(&ill->ill_lock);
12870         ill_dlpi_dispatch(ill, mp);
12871 }

12873 void
12874 ill_capability_send(ill_t *ill, mblk_t *mp)
12875 {
12876         ill->ill_capab_pending_cnt++;
12877         ill_dlpi_send(ill, mp);
12878 }

12880 void
12881 ill_capability_done(ill_t *ill)
12882 {
12883         ASSERT(ill->ill_capab_pending_cnt != 0);

12885         ill_dlpi_done(ill, DL_CAPABILITY_REQ);

12887         ill->ill_capab_pending_cnt--;
12888         if (ill->ill_capab_pending_cnt == 0 &&
12889             ill->ill_dlpi_capab_state == IDCS_OK)
12890                 ill_capability_reset_alloc(ill);
12891 }

12893 /*
12894  * Send all deferred DLPI messages without waiting for their ACKs.
12895  */
12896 void
12897 ill_dlpi_send_deferred(ill_t *ill)
12898 {
12899         mblk_t *mp, *nextmp;

12901         /*
12902          * Clear ill_dlpi_pending so that the message is not queued in
12903          * ill_dlpi_send().
12904          */
12905         mutex_enter(&ill->ill_lock);
12906         ill->ill_dlpi_pending = DL_PRIM_INVAL;
12907         mp = ill->ill_dlpi_deferred;
12908         ill->ill_dlpi_deferred = NULL;
12909         mutex_exit(&ill->ill_lock);

12911         for (; mp != NULL; mp = nextmp) {
12912                 nextmp = mp->b_next;
12913                 mp->b_next = NULL;
12914                 ill_dlpi_send(ill, mp);
12915         }
12916 }

12918 /*
12919  * Clear all the deferred DLPI messages. Called on receiving an M_ERROR
12920  * or M_HANGUP
12921  */
12922 static void
12923 ill_dlpi_clear_deferred(ill_t *ill)
12924 {
12925         mblk_t  *mp, *nextmp;

12927         mutex_enter(&ill->ill_lock);
12928         ill->ill_dlpi_pending = DL_PRIM_INVAL;
12929         mp = ill->ill_dlpi_deferred;
12930         ill->ill_dlpi_deferred = NULL;
12931         mutex_exit(&ill->ill_lock);
```

```
12933         for (; mp != NULL; mp = nextmp) {
12934                 nextmp = mp->b_next;
12935                 inet_freemsg(mp);
12936         }
12937 }

12939 /*
12940  * Check if the DLPI primitive 'prim' is pending; print a warning if not.
12941  */
12942 boolean_t
12943 ill_dlpi_pending(ill_t *ill, t_uscalar_t prim)
12944 {
12945         t_uscalar_t pending;

12947         mutex_enter(&ill->ill_lock);
12948         if (ill->ill_dlpi_pending == prim) {
12949                 mutex_exit(&ill->ill_lock);
12950                 return (B_TRUE);
12951         }

12953         /*
12954          * During teardown, ill_dlpi_dispatch() will send DLPI requests
12955          * without waiting, so don't print any warnings in that case.
12956          */
12957         if (ill->ill_state_flags & ILL_CONDEMNED) {
12958                 mutex_exit(&ill->ill_lock);
12959                 return (B_FALSE);
12960         }
12961         pending = ill->ill_dlpi_pending;
12962         mutex_exit(&ill->ill_lock);

12964         if (pending == DL_PRIM_INVAL) {
12965                 (void) mi_strlog(ill->ill_rq, 1, SL_CONSOLE|SL_ERROR|SL_TRACE,
12966                     "received unsolicited ack for %s on %s\n",
12967                     dl_primstr(prim), ill->ill_name);
12968         } else {
12969                 (void) mi_strlog(ill->ill_rq, 1, SL_CONSOLE|SL_ERROR|SL_TRACE,
12970                     "received unexpected ack for %s on %s (expecting %s)\n",
12971                     dl_primstr(prim), ill->ill_name, dl_primstr(pending));
12972         }
12973         return (B_FALSE);
12974 }

12976 /*
12977  * Complete the current DLPI operation associated with 'prim' on 'ill' and
12978  * start the next queued DLPI operation (if any).  If there are no queued DLPI
12979  * operations and the ill's current exclusive IPSQ operation has finished
12980  * (i.e., ipsq_current_finish() was called), then clear ipsq_current_ipif to
12981  * allow the next exclusive IPSQ operation to begin upon ipsq_exit().  See
12982  * the comments above ipsq_current_finish() for details.
12983  */
12984 void
12985 ill_dlpi_done(ill_t *ill, t_uscalar_t prim)
12986 {
12987         mblk_t *mp;
12988         ipsq_t *ipsq = ill->ill_phyint->phyint_ipsq;
12989         ipxop_t *ipx = ipsq->ipsq_xop;

12991         ASSERT(IAM_WRITER_IPSQ(ipsq));
12992         mutex_enter(&ill->ill_lock);

12994         ASSERT(prim != DL_PRIM_INVAL);
12995         ASSERT(ill->ill_dlpi_pending == prim);

12997         ip1dbg(("ill_dlpi_done: %s has completed %s (%u)\n", ill->ill_name,
```



```
12998             dl_primstr(ill->ill_dlpi_pending), ill->ill_dlpi_pending));

13000         if ((mp = ill->ill_dlpi_deferred) == NULL) {
13001                 ill->ill_dlpi_pending = DL_PRIM_INVAL;
13002                 if (ipx->ipx_current_done) {
13003                         mutex_enter(&ipx->ipx_lock);
13004                         ipx->ipx_current_ipif = NULL;
13005                         mutex_exit(&ipx->ipx_lock);
13006                 }
13007                 cv_signal(&ill->ill_cv);
13008                 mutex_exit(&ill->ill_lock);
13009                 return;
13010         }

13012         ill->ill_dlpi_deferred = mp->b_next;
13013         mp->b_next = NULL;
13014         mutex_exit(&ill->ill_lock);

13016         ill_dlpi_dispatch(ill, mp);
13017 }

13019 /*
13020  * Queue a (multicast) DLPI control message to be sent to the driver by
13021  * later calling ill_dlpi_send_queued.
13022  * We queue them while holding a lock (ill_mcast_lock) to ensure that they
13023  * are sent in order i.e., prevent a DL_DISABMULTI_REQ and DL_ENABMULTI_REQ
13024  * for the same group to race.
13025  * We send DLPI control messages in order using ill_lock.
13026  * For IPMP we should be called on the cast_ill.
13027  */
13028 void
13029 ill_dlpi_queue(ill_t *ill, mblk_t *mp)
13030 {
13031         mblk_t **mpp;

13033         ASSERT(DB_TYPE(mp) == M_PROTO || DB_TYPE(mp) == M_PCPROTO);

13035         mutex_enter(&ill->ill_lock);
13036         /* Must queue message. Tail insertion */
13037         mpp = &ill->ill_dlpi_deferred;
13038         while (*mpp != NULL)
13039                 mpp = &((*mpp)->b_next);

13041         *mpp = mp;
13042         mutex_exit(&ill->ill_lock);
13043 }

13045 /*
13046  * Send the messages that were queued. Make sure there is only
13047  * one outstanding message. ip_rput_dlpi_writer calls ill_dlpi_done()
13048  * when an ACK or a NAK is received to process the next queued message.
13049  * For IPMP we are called on the upper ill, but when send what is queued
13050  * on the cast_ill.
13051  */
13052 void
13053 ill_dlpi_send_queued(ill_t *ill)
13054 {
13055         mblk_t  *mp;
13056         union DL_primitives *dlp;
13057         t_uscalar_t prim;
13058         ill_t *release_ill = NULL;

13060         if (IS_IPMP(ill)) {
13061                 /* On the upper IPMP ill. */
13062                 release_ill = ipmp_illgrp_hold_cast_ill(ill->ill_grp);
13063                 if (release_ill == NULL) {
```

```
13064                         /* Avoid ever sending anything down to the ipmpstub */
13065                         return;
13066                 }
13067                 ill = release_ill;
13068         }
13069         mutex_enter(&ill->ill_lock);
13070         while ((mp = ill->ill_dlpi_deferred) != NULL) {
13071                 if (ill->ill_dlpi_pending != DL_PRIM_INVAL) {
13072                         /* Can't send. Somebody else will send it */
13073                         mutex_exit(&ill->ill_lock);
13074                         goto done;
13075                 }
13076                 ill->ill_dlpi_deferred = mp->b_next;
13077                 mp->b_next = NULL;
13078                 if (!ill->ill_dl_up) {
13079                         /*
13080                          * Nobody there. All multicast addresses will be
13081                          * re-joined when we get the DL_BIND_ACK bringing the
13082                          * interface up.
13083                          */
13084                         freemsg(mp);
13085                         continue;
13086                 }
13087                 dlp = (union DL_primitives *)mp->b_rptr;
13088                 prim = dlp->dl_primitive;

13090                 if (!(ill->ill_state_flags & ILL_CONDEMNED) ||
13091                     (prim == DL_UNBIND_REQ)) {
13092                         ill->ill_dlpi_pending = prim;
13093                 }
13094                 mutex_exit(&ill->ill_lock);

13096                 DTRACE_PROBE3(ill__dlpi, char *, "ill_dlpi_send_queued",
13097                     char *, dl_primstr(prim), ill_t *, ill);
13098                 putnext(ill->ill_wq, mp);
13099                 mutex_enter(&ill->ill_lock);
13100         }
13101         mutex_exit(&ill->ill_lock);
13102 done:
13103         if (release_ill != NULL)
13104                 ill_refrele(release_ill);
13105 }

13107 /*
13108  * Queue an IP (IGMP/MLD) message to be sent by IP from
13109  * ill_mcast_send_queued
13110  * We queue them while holding a lock (ill_mcast_lock) to ensure that they
13111  * are sent in order i.e., prevent a IGMP leave and IGMP join for the same
13112  * group to race.
13113  * We send them in order using ill_lock.
13114  * For IPMP we are called on the upper ill, but we queue on the cast_ill.
13115  */
13116 void
13117 ill_mcast_queue(ill_t *ill, mblk_t *mp)
13118 {
13119         mblk_t **mpp;
13120         ill_t *release_ill = NULL;

13122         ASSERT(RW_LOCK_HELD(&ill->ill_mcast_lock));

13124         if (IS_IPMP(ill)) {
13125                 /* On the upper IPMP ill. */
13126                 release_ill = ipmp_illgrp_hold_cast_ill(ill->ill_grp);
13127                 if (release_ill == NULL) {
13128                         /* Discard instead of queuing for the ipmp interface */
13129                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
```



```
13130                         ip_drop_output("ipIfStatsOutDiscards - no cast_ill",
13131                             mp, ill);
13132                         freemsg(mp);
13133                         return;
13134                 }
13135                 ill = release_ill;
13136         }

13138         mutex_enter(&ill->ill_lock);
13139         /* Must queue message. Tail insertion */
13140         mpp = &ill->ill_mcast_deferred;
13141         while (*mpp != NULL)
13142                 mpp = &((*mpp)->b_next);

13144         *mpp = mp;
13145         mutex_exit(&ill->ill_lock);
13146         if (release_ill != NULL)
13147                 ill_refrele(release_ill);
13148 }

13150 /*
13151  * Send the IP packets that were queued by ill_mcast_queue.
13152  * These are IGMP/MLD packets.
13153  *
13154  * For IPMP we are called on the upper ill, but when send what is queued
13155  * on the cast_ill.
13156  *
13157  * Request loopback of the report if we are acting as a multicast
13158  * router, so that the process-level routing demon can hear it.
13159  * This will run multiple times for the same group if there are members
13160  * on the same group for multiple ipif's on the same ill. The
13161  * igmp_input/mld_input code will suppress this due to the loopback thus we
13162  * always loopback membership report.
13163  *
13164  * We also need to make sure that this does not get load balanced
13165  * by IPMP. We do this by passing an ill to ip_output_simple.
13166  */
13167 void
13168 ill_mcast_send_queued(ill_t *ill)
13169 {
13170         mblk_t  *mp;
13171         ip_xmit_attr_t ixas;
13172         ill_t *release_ill = NULL;

13174         if (IS_IPMP(ill)) {
13175                 /* On the upper IPMP ill. */
13176                 release_ill = ipmp_illgrp_hold_cast_ill(ill->ill_grp);
13177                 if (release_ill == NULL) {
13178                         /*
13179                          * We should have no messages on the ipmp interface
13180                          * but no point in trying to send them.
13181                          */
13182                         return;
13183                 }
13184                 ill = release_ill;
13185         }
13186         bzero(&ixas, sizeof (ixas));
13187         ixas.ixa_zoneid = ALL_ZONES;
13188         ixas.ixa_cred = kcred;
13189         ixas.ixa_cpid = NOPID;
13190         ixas.ixa_tsl = NULL;
13191         /*
13192          * Here we set ixa_ifindex. If IPMP it will be the lower ill which
13193          * makes ip_select_route pick the IRE_MULTICAST for the cast_ill.
13194          * That is necessary to handle IGMP/MLD snooping switches.
13195          */
```

```
13196         ixas.ixa_ifindex = ill->ill_phyint->phyint_ifindex;
13197         ixas.ixa_ipst = ill->ill_ipst;

13199         mutex_enter(&ill->ill_lock);
13200         while ((mp = ill->ill_mcast_deferred) != NULL) {
13201                 ill->ill_mcast_deferred = mp->b_next;
13202                 mp->b_next = NULL;
13203                 if (!ill->ill_dl_up) {
13204                         /*
13205                          * Nobody there. Just drop the ip packets.
13206                          * IGMP/MLD will resend later, if this is a replumb.
13207                          */
13208                         freemsg(mp);
13209                         continue;
13210                 }
13211                 mutex_enter(&ill->ill_phyint->phyint_lock);
13212                 if (IS_UNDER_IPMP(ill) && !ipmp_ill_is_active(ill)) {
13213                         /*
13214                          * When the ill is getting deactivated, we only want to
13215                          * send the DLPI messages, so drop IGMP/MLD packets.
13216                          * DLPI messages are handled by ill_dlpi_send_queued()
13217                          */
13218                         mutex_exit(&ill->ill_phyint->phyint_lock);
13219                         freemsg(mp);
13220                         continue;
13221                 }
13222                 mutex_exit(&ill->ill_phyint->phyint_lock);
13223                 mutex_exit(&ill->ill_lock);

13225                 /* Check whether we are sending IPv4 or IPv6. */
13226                 if (ill->ill_isv6) {
13227                         ip6_t  *ip6h = (ip6_t *)mp->b_rptr;

13229                         ixas.ixa_multicast_ttl = ip6h->ip6_hops;
13230                         ixas.ixa_flags = IXAF_BASIC_SIMPLE_V6;
13231                 } else {
13232                         ipha_t *ipha = (ipha_t *)mp->b_rptr;

13234                         ixas.ixa_multicast_ttl = ipha->ipha_ttl;
13235                         ixas.ixa_flags = IXAF_BASIC_SIMPLE_V4;
13236                         ixas.ixa_flags &= ~IXAF_SET_ULP_CKSUM;
13237                 }
13238                 ixas.ixa_flags &= ~IXAF_VERIFY_SOURCE;
13239                 ixas.ixa_flags |= IXAF_MULTICAST_LOOP | IXAF_SET_SOURCE;
13240                 (void) ip_output_simple(mp, &ixas);
13241                 ixa_cleanup(&ixas);

13243                 mutex_enter(&ill->ill_lock);
13244         }
13245         mutex_exit(&ill->ill_lock);

13247 done:
13248         if (release_ill != NULL)
13249                 ill_refrele(release_ill);
13250 }

13252 /*
13253  * Take down a specific interface, but don't lose any information about it.
13254  * (Always called as writer.)
13255  * This function goes through the down sequence even if the interface is
13256  * already down. There are 2 reasons.
13257  * a. Currently we permit interface routes that depend on down interfaces
13258  *    to be added. This behaviour itself is questionable. However it appears
13259  *    that both Solaris and 4.3 BSD have exhibited this behaviour for a long
13260  *    time. We go thru the cleanup in order to remove these routes.
13261  * b. The bringup of the interface could fail in ill_dl_up i.e. we get
```



```
13262  *    DL_ERROR_ACK in response to the DL_BIND request. The interface is
13263  *    down, but we need to cleanup i.e. do ill_dl_down and
13264  *    ip_rput_dlpi_writer (DL_ERROR_ACK) -> ipif_down.
13265  *
13266  * IP-MT notes:
13267  *
13268  * Model of reference to interfaces.
13269  *
13270  * The following members in ipif_t track references to the ipif.
13271  *      int     ipif_refcnt;    Active reference count
13272  *
13273  * The following members in ill_t track references to the ill.
13274  *      int             ill_refcnt;     active refcnt
13275  *      uint_t          ill_ire_cnt;    Number of ires referencing ill
13276  *      uint_t          ill_ncec_cnt;   Number of ncecs referencing ill
13277  *      uint_t          ill_nce_cnt;    Number of nces referencing ill
13278  *      uint_t          ill_ilm_cnt;    Number of ilms referencing ill
13279  *
13280  * Reference to an ipif or ill can be obtained in any of the following ways.
13281  *
13282  * Through the lookup functions ipif_lookup_* / ill_lookup_* functions
13283  * Pointers to ipif / ill from other data structures viz ire and conn.
13284  * Implicit reference to the ipif / ill by holding a reference to the ire.
13285  *
13286  * The ipif/ill lookup functions return a reference held ipif / ill.
13287  * ipif_refcnt and ill_refcnt track the reference counts respectively.
13288  * This is a purely dynamic reference count associated with threads holding
13289  * references to the ipif / ill. Pointers from other structures do not
13290  * count towards this reference count.
13291  *
13292  * ill_ire_cnt is the number of ire's associated with the
13293  * ill. This is incremented whenever a new ire is created referencing the
13294  * ill. This is done atomically inside ire_add_v[46] where the ire is
13295  * actually added to the ire hash table. The count is decremented in
13296  * ire_inactive where the ire is destroyed.
13297  *
13298  * ill_ncec_cnt is the number of ncec's referencing the ill thru ncec_ill.
13299  * This is incremented atomically in
13300  * ndp_add_v4()/ndp_add_v6() where the nce is actually added to the
13301  * table. Similarly it is decremented in ncec_inactive() where the ncec
13302  * is destroyed.
13303  *
13304  * ill_nce_cnt is the number of nce's referencing the ill thru nce_ill. This is
13305  * incremented atomically in nce_add() where the nce is actually added to the
13306  * ill_nce. Similarly it is decremented in nce_inactive() where the nce
13307  * is destroyed.
13308  *
13309  * ill_ilm_cnt is the ilm's reference to the ill. It is incremented in
13310  * ilm_add() and decremented before the ilm is freed in ilm_delete().
13311  *
13312  * Flow of ioctls involving interface down/up
13313  *
13314  * The following is the sequence of an attempt to set some critical flags on an
13315  * up interface.
13316  * ip_sioctl_flags
13317  * ipif_down
13318  * wait for ipif to be quiescent
13319  * ipif_down_tail
13320  * ip_sioctl_flags_tail
13321  *
13322  * All set ioctls that involve down/up sequence would have a skeleton similar
13323  * to the above. All the *tail functions are called after the refcounts have
13324  * dropped to the appropriate values.
13325  *
13326  * SIOC ioctls during the IPIF_CHANGING interval.
13327  *
```

```
13328  * Threads handling SIOC set ioctls serialize on the squeue, but this
13329  * is not done for SIOC get ioctls. Since a set ioctl can cause several
13330  * steps of internal changes to the state, some of which are visible in
13331  * ipif_flags (such as IFF_UP being cleared and later set), and we want
13332  * the set ioctl to be atomic related to the get ioctls, the SIOC get code
13333  * will wait and restart ioctls if IPIF_CHANGING is set. The mblk is then
13334  * enqueued in the ipsq and the operation is restarted by ipsq_exit() when
13335  * the current exclusive operation completes. The IPIF_CHANGING check
13336  * and enqueue is atomic using the ill_lock and ipsq_lock. The
13337  * lookup is done holding the ill_lock. Hence the ill/ipif state flags can't
13338  * change while the ill_lock is held. Before dropping the ill_lock we acquire
13339  * the ipsq_lock and call ipsq_enq. This ensures that ipsq_exit can't finish
13340  * until we release the ipsq_lock, even though the ill/ipif state flags
13341  * can change after we drop the ill_lock.
13342  */
13343 int
13344 ipif_down(ipif_t *ipif, queue_t *q, mblk_t *mp)
13345 {
13346         ill_t           *ill = ipif->ipif_ill;
13347         conn_t          *connp;
13348         boolean_t       success;
13349         boolean_t       ipif_was_up = B_FALSE;
13350         ip_stack_t      *ipst = ill->ill_ipst;

13352         ASSERT(IAM_WRITER_IPIF(ipif));

13354         ip1dbg(("ipif_down(%s:%u)\n", ill->ill_name, ipif->ipif_id));

13356         DTRACE_PROBE3(ipif__downup, char *, "ipif_down",
13357             ill_t *, ill, ipif_t *, ipif);

13359         if (ipif->ipif_flags & IPIF_UP) {
13360                 mutex_enter(&ill->ill_lock);
13361                 ipif->ipif_flags &= ~IPIF_UP;
13362                 ASSERT(ill->ill_ipif_up_count > 0);
13363                 --ill->ill_ipif_up_count;
13364                 mutex_exit(&ill->ill_lock);
13365                 ipif_was_up = B_TRUE;
13366                 /* Update status in SCTP's list */
13367                 sctp_update_ipif(ipif, SCTP_IPIF_DOWN);
13368                 ill_nic_event_dispatch(ipif->ipif_ill,
13369                     MAP_IPIF_ID(ipif->ipif_id), NE_LIF_DOWN, NULL, 0);
13370         }

13372         /*
13373          * Removal of the last ipif from an ill may result in a DL_UNBIND
13374          * being sent to the driver, and we must not send any data packets to
13375          * the driver after the DL_UNBIND_REQ. To ensure this, all the
13376          * ire and nce entries used in the data path will be cleaned
13377          * up, and we also set  the ILL_DOWN_IN_PROGRESS bit to make
13378          * sure on new entries will be added until the ill is bound
13379          * again. The ILL_DOWN_IN_PROGRESS bit is turned off upon
13380          * receipt of a DL_BIND_ACK.
13381          */
13382         if (ill->ill_wq != NULL && !ill->ill_logical_down &&
13383             ill->ill_ipif_up_count == 0 && ill->ill_ipif_dup_count == 0 &&
13384             ill->ill_dl_up) {
13385                 ill->ill_state_flags |= ILL_DOWN_IN_PROGRESS;
13386         }

13388         /*
13389          * Blow away memberships we established in ipif_multicast_up().
13390          */
13391         ipif_multicast_down(ipif);

13393         /*
```



```
13394          * Remove from the mapping for __sin6_src_id. We insert only
13395          * when the address is not INADDR_ANY. As IPv4 addresses are
13396          * stored as mapped addresses, we need to check for mapped
13397          * INADDR_ANY also.
13398          */
13399         if (ipif_was_up && !IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6lcl_addr) &&
13400             !IN6_IS_ADDR_V4MAPPED_ANY(&ipif->ipif_v6lcl_addr) &&
13401             !(ipif->ipif_flags & IPIF_NOLOCAL)) {
13402                 int err;

13404                 err = ip_srcid_remove(&ipif->ipif_v6lcl_addr,
13405                     ipif->ipif_zoneid, ipst);
13406                 if (err != 0) {
13407                         ip0dbg(("ipif_down: srcid_remove %d\n", err));
13408                 }
13409         }

13411         if (ipif_was_up) {
13412                 /* only delete if we'd added ire's before */
13413                 if (ipif->ipif_isv6)
13414                         ipif_delete_ires_v6(ipif);
13415                 else
13416                         ipif_delete_ires_v4(ipif);
13417         }

13419         if (ipif_was_up && ill->ill_ipif_up_count == 0) {
13420                 /*
13421                  * Since the interface is now down, it may have just become
13422                  * inactive.  Note that this needs to be done even for a
13423                  * lll_logical_down(), or ARP entries will not get correctly
13424                  * restored when the interface comes back up.
13425                  */
13426                 if (IS_UNDER_IPMP(ill))
13427                         ipmp_ill_refresh_active(ill);
13428         }

13430         /*
13431          * neighbor-discovery or arp entries for this interface. The ipif
13432          * has to be quiesced, so we walk all the nce's and delete those
13433          * that point at the ipif->ipif_ill. At the same time, we also
13434          * update IPMP so that ipifs for data addresses are unbound. We dont
13435          * call ipif_arp_down to DL_UNBIND the arp stream itself here, but defer
13436          * that for ipif_down_tail()
13437          */
13438         ipif_nce_down(ipif);

13440         /*
13441          * If this is the last ipif on the ill, we also need to remove
13442          * any IREs with ire_ill set. Otherwise ipif_is_quiescent() will
13443          * never succeed.
13444          */
13445         if (ill->ill_ipif_up_count == 0 && ill->ill_ipif_dup_count == 0)
13446                 ire_walk_ill(0, 0, ill_downi, ill, ill);

13448         /*
13449          * Walk all CONNs that can have a reference on an ire for this
13450          * ipif (we actually walk all that now have stale references).
13451          */
13452         ipcl_walk(conn_ixa_cleanup, (void *)B_TRUE, ipst);

13454         /*
13455          * If mp is NULL the caller will wait for the appropriate refcnt.
13456          * Eg. ip_sioctl_removeif -> ipif_free  -> ipif_down
13457          * and ill_delete -> ipif_free -> ipif_down
13458          */
13459         if (mp == NULL) {
```

```
13460                   ASSERT(q == NULL);
13461                   return (0);
13462           }

13464           if (CONN_Q(q)) {
13465                   connp = Q_TO_CONN(q);
13466                   mutex_enter(&connp->conn_lock);
13467           } else {
13468                   connp = NULL;
13469           }
13470           mutex_enter(&ill->ill_lock);
13471           /*
13472            * Are there any ire's pointing to this ipif that are still active ?
13473            * If this is the last ipif going down, are there any ire's pointing
13474            * to this ill that are still active ?
13475            */
13476           if (ipif_is_quiescent(ipif)) {
13477                   mutex_exit(&ill->ill_lock);
13478                   if (connp != NULL)
13479                           mutex_exit(&connp->conn_lock);
13480                   return (0);
13481           }

13483           ip1dbg(("ipif_down: need to wait, adding pending mp %s ill %p",
13484               ill->ill_name, (void *)ill));
13485           /*
13486            * Enqueue the mp atomically in ipsq_pending_mp. When the refcount
13487            * drops down, the operation will be restarted by ipif_ill_refrele_tail
13488            * which in turn is called by the last refrele on the ipif/ill/ire.
13489            */
13490           success = ipsq_pending_mp_add(connp, ipif, q, mp, IPIF_DOWN);
13491           if (!success) {
13492                   /* The conn is closing. So just return */
13493                   ASSERT(connp != NULL);
13494                   mutex_exit(&ill->ill_lock);
13495                   mutex_exit(&connp->conn_lock);
13496                   return (EINTR);
13497           }

13499           mutex_exit(&ill->ill_lock);
13500           if (connp != NULL)
13501                   mutex_exit(&connp->conn_lock);
13502           return (EINPROGRESS);
13503   }

13505   int
13506   ipif_down_tail(ipif_t *ipif)
13507   {
13508           ill_t   *ill = ipif->ipif_ill;
13509           int     err = 0;

13511           DTRACE_PROBE3(ipif__downup, char *, "ipif_down_tail",
13512               ill_t *, ill, ipif_t *, ipif);

13514           /*
13515            * Skip any loopback interface (null wq).
13516            * If this is the last logical interface on the ill
13517            * have ill_dl_down tell the driver we are gone (unbind)
13518            * Note that lun 0 can ipif_down even though
13519            * there are other logical units that are up.
13520            * This occurs e.g. when we change a "significant" IFF_ flag.
13521            */
13522           if (ill->ill_wq != NULL && !ill->ill_logical_down &&
13523               ill->ill_ipif_up_count == 0 && ill->ill_ipif_dup_count == 0 &&
13524               ill->ill_dl_up) {
13525                   ill_dl_down(ill);
```



```
13526           }
13527           if (!ipif->ipif_isv6)
13528                   err = ipif_arp_down(ipif);

13530           ill->ill_logical_down = 0;

13532           ip_rts_ifmsg(ipif, RTSQ_DEFAULT);
13533           ip_rts_newaddrmsg(RTM_DELETE, 0, ipif, RTSQ_DEFAULT);
13534           return (err);
13535   }

13537   /*
13538    * Bring interface logically down without bringing the physical interface
13539    * down e.g. when the netmask is changed. This avoids long lasting link
13540    * negotiations between an ethernet interface and a certain switches.
13541    */
13542   static int
13543   ipif_logical_down(ipif_t *ipif, queue_t *q, mblk_t *mp)
13544   {
13545           DTRACE_PROBE3(ipif__downup, char *, "ipif_logical_down",
13546               ill_t *, ipif->ipif_ill, ipif_t *, ipif);

13548           /*
13549            * The ill_logical_down flag is a transient flag. It is set here
13550            * and is cleared once the down has completed in ipif_down_tail.
13551            * This flag does not indicate whether the ill stream is in the
13552            * DL_BOUND state with the driver. Instead this flag is used by
13553            * ipif_down_tail to determine whether to DL_UNBIND the stream with
13554            * the driver. The state of the ill stream i.e. whether it is
13555            * DL_BOUND with the driver or not is indicated by the ill_dl_up flag.
13556            */
13557           ipif->ipif_ill->ill_logical_down = 1;
13558           return (ipif_down(ipif, q, mp));
13559   }

13561   /*
13562    * Initiate deallocate of an IPIF. Always called as writer. Called by
13563    * ill_delete or ip_sioctl_removeif.
13564    */
13565   static void
13566   ipif_free(ipif_t *ipif)
13567   {
13568           ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;

13570           ASSERT(IAM_WRITER_IPIF(ipif));

13572           if (ipif->ipif_recovery_id != 0)
13573                   (void) untimeout(ipif->ipif_recovery_id);
13574           ipif->ipif_recovery_id = 0;

13576           /*
13577            * Take down the interface. We can be called either from ill_delete
13578            * or from ip_sioctl_removeif.
13579            */
13580           (void) ipif_down(ipif, NULL, NULL);

13582           /*
13583            * Now that the interface is down, there's no chance it can still
13584            * become a duplicate.  Cancel any timer that may have been set while
13585            * tearing down.
13586            */
13587           if (ipif->ipif_recovery_id != 0)
13588                   (void) untimeout(ipif->ipif_recovery_id);
13589           ipif->ipif_recovery_id = 0;

13591           rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
```

```
13592         /* Remove pointers to this ill in the multicast routing tables */
13593         reset_mrt_vif_ipif(ipif);
13594         /* If necessary, clear the cached source ipif rotor. */
13595         if (ipif->ipif_ill->ill_src_ipif == ipif)
13596                 ipif->ipif_ill->ill_src_ipif = NULL;
13597         rw_exit(&ipst->ips_ill_g_lock);
13598 }

13600 static void
13601 ipif_free_tail(ipif_t *ipif)
13602 {
13603         ip_stack_t *ipst = ipif->ipif_ill->ill_ipst;

13605         /*
13606          * Need to hold both ill_g_lock and ill_lock while
13607          * inserting or removing an ipif from the linked list
13608          * of ipifs hanging off the ill.
13609          */
13610         rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);

13612 #ifdef DEBUG
13613         ipif_trace_cleanup(ipif);
13614 #endif

13616         /* Ask SCTP to take it out of it list */
13617         sctp_update_ipif(ipif, SCTP_IPIF_REMOVE);
13618         ip_rts_newaddrmsg(RTM_FREEADDR, 0, ipif, RTSQ_DEFAULT);

13620         /* Get it out of the ILL interface list. */
13621         ipif_remove(ipif);
13622         rw_exit(&ipst->ips_ill_g_lock);

13624         ASSERT(!(ipif->ipif_flags & (IPIF_UP | IPIF_DUPLICATE)));
13625         ASSERT(ipif->ipif_recovery_id == 0);
13626         ASSERT(ipif->ipif_ire_local == NULL);
13627         ASSERT(ipif->ipif_ire_if == NULL);

13629         /* Free the memory. */
13630         mi_free(ipif);
13631 }

13633 /*
13634  * Sets ‘buf’ to an ipif name of the form "ill_name:id", or "ill_name" if "id"
13635  * is zero.
13636  */
13637 void
13638 ipif_get_name(const ipif_t *ipif, char *buf, int len)
13639 {
13640         char    lbuf[LIFNAMSIZ];
13641         char    *name;
13642         size_t  name_len;

13644         buf[0] = ’\0’;
13645         name = ipif->ipif_ill->ill_name;
13646         name_len = ipif->ipif_ill->ill_name_length;
13647         if (ipif->ipif_id != 0) {
13648                 (void) sprintf(lbuf, "%s%c%d", name, IPIF_SEPARATOR_CHAR,
13649                     ipif->ipif_id);
13650                 name = lbuf;
13651                 name_len = mi_strlen(name) + 1;
13652         }
13653         len -= 1;
13654         buf[len] = ’\0’;
13655         len = MIN(len, name_len);
13656         bcopy(name, buf, len);
13657 }
```



```
13659 /*
13660  * Sets ‘buf’ to an ill name.
13661  */
13662 void
13663 ill_get_name(const ill_t *ill, char *buf, int len)
13664 {
13665         char    *name;
13666         size_t  name_len;

13668         name = ill->ill_name;
13669         name_len = ill->ill_name_length;
13670         len -= 1;
13671         buf[len] = ’\0’;
13672         len = MIN(len, name_len);
13673         bcopy(name, buf, len);
13674 }

13676 /*
13677  * Find an IPIF based on the name passed in.  Names can be of the form <phys>
13678  * (e.g., le0) or <phys>:<#> (e.g., le0:1).  When there is no colon, the
13679  * implied unit id is zero. <phys> must correspond to the name of an ILL.
13680  * (May be called as writer.)
13681  */
13682 static ipif_t *
13683 ipif_lookup_on_name(char *name, size_t namelen, boolean_t do_alloc,
13684     boolean_t *exists, boolean_t isv6, zoneid_t zoneid, ip_stack_t *ipst)
13685 {
13686         char    *cp;
13687         char    *endp;
13688         long    id;
13689         ill_t   *ill;
13690         ipif_t  *ipif;
13691         uint_t  ire_type;
13692         boolean_t did_alloc = B_FALSE;
13693         char    last;

13695         /*
13696          * If the caller wants to us to create the ipif, make sure we have a
13697          * valid zoneid
13698          */
13699         ASSERT(!do_alloc || zoneid != ALL_ZONES);

13701         if (namelen == 0) {
13702                 return (NULL);
13703         }

13705         *exists = B_FALSE;
13706         /* Look for a colon in the name. */
13707         endp = &name[namelen];
13708         for (cp = endp; --cp > name; ) {
13709                 if (*cp == IPIF_SEPARATOR_CHAR)
13710                         break;
13711         }

13713         if (*cp == IPIF_SEPARATOR_CHAR) {
13714                 /*
13715                  * Reject any non-decimal aliases for logical
13716                  * interfaces. Aliases with leading zeroes
13717                  * are also rejected as they introduce ambiguity
13718                  * in the naming of the interfaces.
13719                  * In order to confirm with existing semantics,
13720                  * and to not break any programs/script relying
13721                  * on that behaviour, if<0>:0 is considered to be
13722                  * a valid interface.
13723                  *
```

```
13724                  * If alias has two or more digits and the first
13725                  * is zero, fail.
13726                  */
13727                 if (&cp[2] < endp && cp[1] == '0') {
13728                         return (NULL);
13729                 }
13730         }

13732         if (cp <= name) {
13733                 cp = endp;
13734         }
13735         last = *cp;
13736         *cp = '\0';

13738         /*
13739          * Look up the ILL, based on the portion of the name
13740          * before the slash. ill_lookup_on_name returns a held ill.
13741          * Temporary to check whether ill exists already. If so
13742          * ill_lookup_on_name will clear it.
13743          */
13744         ill = ill_lookup_on_name(name, do_alloc, isv6,
13745             &did_alloc, ipst);
13746         *cp = last;
13747         if (ill == NULL)
13748                 return (NULL);

13750         /* Establish the unit number in the name. */
13751         id = 0;
13752         if (cp < endp && *endp == '\0') {
13753                 /* If there was a colon, the unit number follows. */
13754                 cp++;
13755                 if (ddi_strtol(cp, NULL, 0, &id) != 0) {
13756                         ill_refrele(ill);
13757                         return (NULL);
13758                 }
13759         }

13761         mutex_enter(&ill->ill_lock);
13762         /* Now see if there is an IPIF with this unit number. */
13763         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
13764                 if (ipif->ipif_id == id) {
13765                         if (zoneid != ALL_ZONES &&
13766                             zoneid != ipif->ipif_zoneid &&
13767                             ipif->ipif_zoneid != ALL_ZONES) {
13768                                 mutex_exit(&ill->ill_lock);
13769                                 ill_refrele(ill);
13770                                 return (NULL);
13771                         }
13772                         if (IPIF_CAN_LOOKUP(ipif)) {
13773                                 ipif_refhold_locked(ipif);
13774                                 mutex_exit(&ill->ill_lock);
13775                                 if (!did_alloc)
13776                                         *exists = B_TRUE;
13777                                 /*
13778                                  * Drop locks before calling ill_refrele
13779                                  * since it can potentially call into
13780                                  * ipif_ill_refrele_tail which can end up
13781                                  * in trying to acquire any lock.
13782                                  */
13783                                 ill_refrele(ill);
13784                                 return (ipif);
13785                         }
13786                 }
13787         }

13789         if (!do_alloc) {
```



```
13790                 mutex_exit(&ill->ill_lock);
13791                 ill_refrele(ill);
13792                 return (NULL);
13793         }

13795         /*
13796          * If none found, atomically allocate and return a new one.
13797          * Historically, we used IRE_LOOPBACK only for lun 0, and IRE_LOCAL
13798          * to support "receive only" use of lo0:1 etc. as is still done
13799          * below as an initial guess.
13800          * However, this is now likely to be overriden later in ipif_up_done()
13801          * when we know for sure what address has been configured on the
13802          * interface, since we might have more than one loopback interface
13803          * with a loopback address, e.g. in the case of zones, and all the
13804          * interfaces with loopback addresses need to be marked IRE_LOOPBACK.
13805          */
13806         if (ill->ill_net_type == IRE_LOOPBACK && id == 0)
13807                 ire_type = IRE_LOOPBACK;
13808         else
13809                 ire_type = IRE_LOCAL;
13810         ipif = ipif_allocate(ill, id, ire_type, B_TRUE, B_TRUE, NULL);
13811         if (ipif != NULL)
13812                 ipif_refhold_locked(ipif);
13813         mutex_exit(&ill->ill_lock);
13814         ill_refrele(ill);
13815         return (ipif);
13816 }

13818 /*
13819  * Variant of the above that queues the request on the ipsq when
13820  * IPIF_CHANGING is set.
13821  */
13822 static ipif_t *
13823 ipif_lookup_on_name_async(char *name, size_t namelen, boolean_t isv6,
13824     zoneid_t zoneid, queue_t *q, mblk_t *mp, ipsq_func_t func, int *error,
13825     ip_stack_t *ipst)
13826 {
13827         char    *cp;
13828         char    *endp;
13829         long    id;
13830         ill_t   *ill;
13831         ipif_t  *ipif;
13832         boolean_t did_alloc = B_FALSE;
13833         ipsq_t  *ipsq;

13835         if (error != NULL)
13836                 *error = 0;

13838         if (namelen == 0) {
13839                 if (error != NULL)
13840                         *error = ENXIO;
13841                 return (NULL);
13842         }

13844         /* Look for a colon in the name. */
13845         endp = &name[namelen];
13846         for (cp = endp; --cp > name; ) {
13847                 if (*cp == IPIF_SEPARATOR_CHAR)
13848                         break;
13849         }

13851         if (*cp == IPIF_SEPARATOR_CHAR) {
13852                 /*
13853                  * Reject any non-decimal aliases for logical
13854                  * interfaces. Aliases with leading zeroes
13855                  * are also rejected as they introduce ambiguity
```

```
13856                   * in the naming of the interfaces.
13857                   * In order to confirm with existing semantics,
13858                   * and to not break any programs/script relying
13859                   * on that behaviour, if<0>:0 is considered to be
13860                   * a valid interface.
13861                   *
13862                   * If alias has two or more digits and the first
13863                   * is zero, fail.
13864                   */
13865                  if (&cp[2] < endp && cp[1] == '0') {
13866                          if (error != NULL)
13867                                  *error = EINVAL;
13868                          return (NULL);
13869                  }
13870          }

13872          if (cp <= name) {
13873                  cp = endp;
13874          } else {
13875                  *cp = '\0';
13876          }

13878          /*
13879           * Look up the ILL, based on the portion of the name
13880           * before the slash. ill_lookup_on_name returns a held ill.
13881           * Temporary to check whether ill exists already. If so
13882           * ill_lookup_on_name will clear it.
13883           */
13884          ill = ill_lookup_on_name(name, B_FALSE, isv6, &did_alloc, ipst);
13885          if (cp != endp)
13886                  *cp = IPIF_SEPARATOR_CHAR;
13887          if (ill == NULL)
13888                  return (NULL);

13890          /* Establish the unit number in the name. */
13891          id = 0;
13892          if (cp < endp && *endp == '\0') {
13893                  /* If there was a colon, the unit number follows. */
13894                  cp++;
13895                  if (ddi_strtol(cp, NULL, 0, &id) != 0) {
13896                          ill_refrele(ill);
13897                          if (error != NULL)
13898                                  *error = ENXIO;
13899                          return (NULL);
13900                  }
13901          }

13903          GRAB_CONN_LOCK(q);
13904          mutex_enter(&ill->ill_lock);
13905          /* Now see if there is an IPIF with this unit number. */
13906          for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
13907                  if (ipif->ipif_id == id) {
13908                          if (zoneid != ALL_ZONES &&
13909                              zoneid != ipif->ipif_zoneid &&
13910                              ipif->ipif_zoneid != ALL_ZONES) {
13911                                  mutex_exit(&ill->ill_lock);
13912                                  RELEASE_CONN_LOCK(q);
13913                                  ill_refrele(ill);
13914                                  if (error != NULL)
13915                                          *error = ENXIO;
13916                                  return (NULL);
13917                          }

13919                          if (!(IPIF_IS_CHANGING(ipif) ||
13920                              IPIF_IS_CONDEMNED(ipif)) ||
13921                              IAM_WRITER_IPIF(ipif)) {
```



```
13922                                  ipif_refhold_locked(ipif);
13923                                  mutex_exit(&ill->ill_lock);
13924                                  /*
13925                                   * Drop locks before calling ill_refrele
13926                                   * since it can potentially call into
13927                                   * ipif_ill_refrele_tail which can end up
13928                                   * in trying to acquire any lock.
13929                                   */
13930                                  RELEASE_CONN_LOCK(q);
13931                                  ill_refrele(ill);
13932                                  return (ipif);
13933                          } else if (q != NULL && !IPIF_IS_CONDEMNED(ipif)) {
13934                                  ipsq = ill->ill_phyint->phyint_ipsq;
13935                                  mutex_enter(&ipsq->ipsq_lock);
13936                                  mutex_enter(&ipsq->ipsq_xop->ipx_lock);
13937                                  mutex_exit(&ill->ill_lock);
13938                                  ipsq_enq(ipsq, q, mp, func, NEW_OP, ill);
13939                                  mutex_exit(&ipsq->ipsq_xop->ipx_lock);
13940                                  mutex_exit(&ipsq->ipsq_lock);
13941                                  RELEASE_CONN_LOCK(q);
13942                                  ill_refrele(ill);
13943                                  if (error != NULL)
13944                                          *error = EINPROGRESS;
13945                                  return (NULL);
13946                          }
13947                  }
13948          }
13949          RELEASE_CONN_LOCK(q);
13950          mutex_exit(&ill->ill_lock);
13951          ill_refrele(ill);
13952          if (error != NULL)
13953                  *error = ENXIO;
13954          return (NULL);
13955  }

13957  /*
13958   * This routine is called whenever a new address comes up on an ipif.  If
13959   * we are configured to respond to address mask requests, then we are supposed
13960   * to broadcast an address mask reply at this time.  This routine is also
13961   * called if we are already up, but a netmask change is made.  This is legal
13962   * but might not make the system manager very popular.  (May be called
13963   * as writer.)
13964   */
13965  void
13966  ipif_mask_reply(ipif_t *ipif)
13967  {
13968          icmph_t *icmph;
13969          ipha_t  *ipha;
13970          mblk_t  *mp;
13971          ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;
13972          ip_xmit_attr_t ixas;

13974  #define REPLY_LEN       (sizeof (icmp_ipha) + sizeof (icmph_t) + IP_ADDR_LEN)

13976          if (!ipst->ips_ip_respond_to_address_mask_broadcast)
13977                  return;

13979          /* ICMP mask reply is IPv4 only */
13980          ASSERT(!ipif->ipif_isv6);
13981          /* ICMP mask reply is not for a loopback interface */
13982          ASSERT(ipif->ipif_ill->ill_wq != NULL);

13984          if (ipif->ipif_lcl_addr == INADDR_ANY)
13985                  return;

13987          mp = allocb(REPLY_LEN, BPRI_HI);
```

```
13988         if (mp == NULL)
13989                 return;
13990         mp->b_wptr = mp->b_rptr + REPLY_LEN;

13992         ipha = (ipha_t *)mp->b_rptr;
13993         bzero(ipha, REPLY_LEN);
13994         *ipha = icmp_ipha;
13995         ipha->ipha_ttl = ipst->ips_ip_broadcast_ttl;
13996         ipha->ipha_src = ipif->ipif_lcl_addr;
13997         ipha->ipha_dst = ipif->ipif_brd_addr;
13998         ipha->ipha_length = htons(REPLY_LEN);
13999         ipha->ipha_ident = 0;

14001         icmph = (icmph_t *)&ipha[1];
14002         icmph->icmph_type = ICMP_ADDRESS_MASK_REPLY;
14003         bcopy(&ipif->ipif_net_mask, &icmph[1], IP_ADDR_LEN);
14004         icmph->icmph_checksum = IP_CSUM(mp, sizeof (ipha_t), 0);

14006         bzero(&ixas, sizeof (ixas));
14007         ixas.ixa_flags = IXAF_BASIC_SIMPLE_V4;
14008         ixas.ixa_zoneid = ALL_ZONES;
14009         ixas.ixa_ifindex = 0;
14010         ixas.ixa_ipst = ipst;
14011         ixas.ixa_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;
14012         (void) ip_output_simple(mp, &ixas);
14013         ixa_cleanup(&ixas);
14014 #undef  REPLY_LEN
14015 }

14017 /*
14018  * Join the ipif specific multicast groups.
14019  * Must be called after a mapping has been set up in the resolver.  (Always
14020  * called as writer.)
14021  */
14022 void
14023 ipif_multicast_up(ipif_t *ipif)
14024 {
14025         int err;
14026         ill_t *ill;
14027         ilm_t *ilm;

14029         ASSERT(IAM_WRITER_IPIF(ipif));

14031         ill = ipif->ipif_ill;

14033         ip1dbg(("ipif_multicast_up\n"));
14034         if (!(ill->ill_flags & ILLF_MULTICAST) ||
14035             ipif->ipif_allhosts_ilm != NULL)
14036                 return;

14038         if (ipif->ipif_isv6) {
14039                 in6_addr_t v6allmc = ipv6_all_hosts_mcast;
14040                 in6_addr_t v6solmc = ipv6_solicited_node_mcast;

14042                 v6solmc.s6_addr32[3] |= ipif->ipif_v6lcl_addr.s6_addr32[3];

14044                 if (IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6lcl_addr))
14045                         return;

14047                 ip1dbg(("ipif_multicast_up - addmulti\n"));

14049                 /*
14050                  * Join the all hosts multicast address.  We skip this for
14051                  * underlying IPMP interfaces since they should be invisible.
14052                  */
14053                 if (!IS_UNDER_IPMP(ill)) {
```



```
14054                         ilm = ip_addmulti(&v6allmc, ill, ipif->ipif_zoneid,
14055                             &err);
14056                         if (ilm == NULL) {
14057                                 ASSERT(err != 0);
14058                                 ip0dbg(("ipif_multicast_up: "
14059                                     "all_hosts_mcast failed %d\n", err));
14060                                 return;
14061                         }
14062                         ipif->ipif_allhosts_ilm = ilm;
14063                 }

14065                 /*
14066                  * Enable multicast for the solicited node multicast address.
14067                  * If IPMP we need to put the membership on the upper ill.
14068                  */
14069                 if (!(ipif->ipif_flags & IPIF_NOLOCAL)) {
14070                         ill_t *mcast_ill = NULL;
14071                         boolean_t need_refrele;

14073                         if (IS_UNDER_IPMP(ill) &&
14074                             (mcast_ill = ipmp_ill_hold_ipmp_ill(ill)) != NULL) {
14075                                 need_refrele = B_TRUE;
14076                         } else {
14077                                 mcast_ill = ill;
14078                                 need_refrele = B_FALSE;
14079                         }

14081                         ilm = ip_addmulti(&v6solmc, mcast_ill,
14082                             ipif->ipif_zoneid, &err);
14083                         if (need_refrele)
14084                                 ill_refrele(mcast_ill);

14086                         if (ilm == NULL) {
14087                                 ASSERT(err != 0);
14088                                 ip0dbg(("ipif_multicast_up: solicited MC"
14089                                     " failed %d\n", err));
14090                                 if ((ilm = ipif->ipif_allhosts_ilm) != NULL) {
14091                                         ipif->ipif_allhosts_ilm = NULL;
14092                                         (void) ip_delmulti(ilm);
14093                                 }
14094                                 return;
14095                         }
14096                         ipif->ipif_solmulti_ilm = ilm;
14097                 }
14098         } else {
14099                 in6_addr_t v6group;

14101                 if (ipif->ipif_lcl_addr == INADDR_ANY || IS_UNDER_IPMP(ill))
14102                         return;

14104                 /* Join the all hosts multicast address */
14105                 ip1dbg(("ipif_multicast_up - addmulti\n"));
14106                 IN6_IPADDR_TO_V4MAPPED(htonl(INADDR_ALLHOSTS_GROUP), &v6group);

14108                 ilm = ip_addmulti(&v6group, ill, ipif->ipif_zoneid, &err);
14109                 if (ilm == NULL) {
14110                         ASSERT(err != 0);
14111                         ip0dbg(("ipif_multicast_up: failed %d\n", err));
14112                         return;
14113                 }
14114                 ipif->ipif_allhosts_ilm = ilm;
14115         }
14116 }

14118 /*
14119  * Blow away any multicast groups that we joined in ipif_multicast_up().
```

```
14120  * (ilms from explicit memberships are handled in conn_update_ill.)
14121  */
14122 void
14123 ipif_multicast_down(ipif_t *ipif)
14124 {
14125         ASSERT(IAM_WRITER_IPIF(ipif));

14127         ip1dbg(("ipif_multicast_down\n"));

14129         if (ipif->ipif_allhosts_ilm != NULL) {
14130                 (void) ip_delmulti(ipif->ipif_allhosts_ilm);
14131                 ipif->ipif_allhosts_ilm = NULL;
14132         }
14133         if (ipif->ipif_solmulti_ilm != NULL) {
14134                 (void) ip_delmulti(ipif->ipif_solmulti_ilm);
14135                 ipif->ipif_solmulti_ilm = NULL;
14136         }
14137 }

14139 /*
14140  * Used when an interface comes up to recreate any extra routes on this
14141  * interface.
14142  */
14143 int
14144 ill_recover_saved_ire(ill_t *ill)
14145 {
14146         mblk_t          *mp;
14147         ip_stack_t      *ipst = ill->ill_ipst;

14149         ip1dbg(("ill_recover_saved_ire(%s)", ill->ill_name));

14151         mutex_enter(&ill->ill_saved_ire_lock);
14152         for (mp = ill->ill_saved_ire_mp; mp != NULL; mp = mp->b_cont) {
14153                 ire_t           *ire, *nire;
14154                 ifrt_t          *ifrt;

14156                 ifrt = (ifrt_t *)mp->b_rptr;
14157                 /*
14158                  * Create a copy of the IRE with the saved address and netmask.
14159                  */
14160                 if (ill->ill_isv6) {
14161                         ire = ire_create_v6(
14162                             &ifrt->ifrt_v6addr,
14163                             &ifrt->ifrt_v6mask,
14164                             &ifrt->ifrt_v6gateway_addr,
14165                             ifrt->ifrt_type,
14166                             ill,
14167                             ifrt->ifrt_zoneid,
14168                             ifrt->ifrt_flags,
14169                             NULL,
14170                             ipst);
14171                 } else {
14172                         ire = ire_create(
14173                             (uint8_t *)&ifrt->ifrt_addr,
14174                             (uint8_t *)&ifrt->ifrt_mask,
14175                             (uint8_t *)&ifrt->ifrt_gateway_addr,
14176                             ifrt->ifrt_type,
14177                             ill,
14178                             ifrt->ifrt_zoneid,
14179                             ifrt->ifrt_flags,
14180                             NULL,
14181                             ipst);
14182                 }
14183                 if (ire == NULL) {
14184                         mutex_exit(&ill->ill_saved_ire_lock);
14185                         return (ENOMEM);
```



```
14186                 }

14188                 if (ifrt->ifrt_flags & RTF_SETSRC) {
14189                         if (ill->ill_isv6) {
14190                                 ire->ire_setsrc_addr_v6 =
14191                                     ifrt->ifrt_v6setsrc_addr;
14192                         } else {
14193                                 ire->ire_setsrc_addr = ifrt->ifrt_setsrc_addr;
14194                         }
14195                 }

14197                 /*
14198                  * Some software (for example, GateD and Sun Cluster) attempts
14199                  * to create (what amount to) IRE_PREFIX routes with the
14200                  * loopback address as the gateway.  This is primarily done to
14201                  * set up prefixes with the RTF_REJECT flag set (for example,
14202                  * when generating aggregate routes.)
14203                  *
14204                  * If the IRE type (as defined by ill->ill_net_type) is
14205                  * IRE_LOOPBACK, then we map the request into a
14206                  * IRE_IF_NORESOLVER.
14207                  */
14208                 if (ill->ill_net_type == IRE_LOOPBACK)
14209                         ire->ire_type = IRE_IF_NORESOLVER;

14211                 /*
14212                  * ire held by ire_add, will be refreled' towards the
14213                  * the end of ipif_up_done
14214                  */
14215                 nire = ire_add(ire);
14216                 /*
14217                  * Check if it was a duplicate entry. This handles
14218                  * the case of two racing route adds for the same route
14219                  */
14220                 if (nire == NULL) {
14221                         ip1dbg(("ill_recover_saved_ire: FAILED\n"));
14222                 } else if (nire != ire) {
14223                         ip1dbg(("ill_recover_saved_ire: duplicate ire %p\n",
14224                             (void *)nire));
14225                         ire_delete(nire);
14226                 } else {
14227                         ip1dbg(("ill_recover_saved_ire: added ire %p\n",
14228                             (void *)nire));
14229                 }
14230                 if (nire != NULL)
14231                         ire_refrele(nire);
14232         }
14233         mutex_exit(&ill->ill_saved_ire_lock);
14234         return (0);
14235 }

14237 /*
14238  * Used to set the netmask and broadcast address to default values when the
14239  * interface is brought up.  (Always called as writer.)
14240  */
14241 static void
14242 ipif_set_default(ipif_t *ipif)
14243 {
14244         ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));

14246         if (!ipif->ipif_isv6) {
14247                 /*
14248                  * Interface holds an IPv4 address. Default
14249                  * mask is the natural netmask.
14250                  */
14251                 if (!ipif->ipif_net_mask) {
```

```
14252                           ipaddr_t        v4mask;

14254                           v4mask = ip_net_mask(ipif->ipif_lcl_addr);
14255                           V4MASK_TO_V6(v4mask, ipif->ipif_v6net_mask);
14256                   }
14257                   if (ipif->ipif_flags & IPIF_POINTOPOINT) {
14258                           /* ipif_subnet is ipif_pp_dst_addr for pt-pt */
14259                           ipif->ipif_v6subnet = ipif->ipif_v6pp_dst_addr;
14260                   } else {
14261                           V6_MASK_COPY(ipif->ipif_v6lcl_addr,
14262                               ipif->ipif_v6net_mask, ipif->ipif_v6subnet);
14263                   }
14264                   /*
14265                    * NOTE: SunOS 4.X does this even if the broadcast address
14266                    * has been already set thus we do the same here.
14267                    */
14268                   if (ipif->ipif_flags & IPIF_BROADCAST) {
14269                           ipaddr_t        v4addr;

14271                           v4addr = ipif->ipif_subnet | ~ipif->ipif_net_mask;
14272                           IN6_IPADDR_TO_V4MAPPED(v4addr, &ipif->ipif_v6brd_addr);
14273                   }
14274           } else {
14275                   /*
14276                    * Interface holds an IPv6-only address.  Default
14277                    * mask is all-ones.
14278                    */
14279                   if (IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6net_mask))
14280                           ipif->ipif_v6net_mask = ipv6_all_ones;
14281                   if (ipif->ipif_flags & IPIF_POINTOPOINT) {
14282                           /* ipif_subnet is ipif_pp_dst_addr for pt-pt */
14283                           ipif->ipif_v6subnet = ipif->ipif_v6pp_dst_addr;
14284                   } else {
14285                           V6_MASK_COPY(ipif->ipif_v6lcl_addr,
14286                               ipif->ipif_v6net_mask, ipif->ipif_v6subnet);
14287                   }
14288           }
14289 }

14291 /*
14292  * Return 0 if this address can be used as local address without causing
14293  * duplicate address problems. Otherwise, return EADDRNOTAVAIL if the address
14294  * is already up on a different ill, and EADDRINUSE if it's up on the same ill.
14295  * Note that the same IPv6 link-local address is allowed as long as the ills
14296  * are not on the same link.
14297  */
14298 int
14299 ip_addr_availability_check(ipif_t *new_ipif)
14300 {
14301         in6_addr_t our_v6addr;
14302         ill_t *ill;
14303         ipif_t *ipif;
14304         ill_walk_context_t ctx;
14305         ip_stack_t      *ipst = new_ipif->ipif_ill->ill_ipst;

14307         ASSERT(IAM_WRITER_IPIF(new_ipif));
14308         ASSERT(MUTEX_HELD(&ipst->ips_ip_addr_avail_lock));
14309         ASSERT(RW_READ_HELD(&ipst->ips_ill_g_lock));

14311         new_ipif->ipif_flags &= ~IPIF_UNNUMBERED;
14312         if (IN6_IS_ADDR_UNSPECIFIED(&new_ipif->ipif_v6lcl_addr) ||
14313             IN6_IS_ADDR_V4MAPPED_ANY(&new_ipif->ipif_v6lcl_addr))
14314                 return (0);

14316         our_v6addr = new_ipif->ipif_v6lcl_addr;
```



```
14318         if (new_ipif->ipif_isv6)
14319                 ill = ILL_START_WALK_V6(&ctx, ipst);
14320         else
14321                 ill = ILL_START_WALK_V4(&ctx, ipst);

14323         for (; ill != NULL; ill = ill_next(&ctx, ill)) {
14324                 for (ipif = ill->ill_ipif; ipif != NULL;
14325                     ipif = ipif->ipif_next) {
14326                         if ((ipif == new_ipif) ||
14327                             !(ipif->ipif_flags & IPIF_UP) ||
14328                             (ipif->ipif_flags & IPIF_UNNUMBERED) ||
14329                             !IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6lcl_addr,
14330                             &our_v6addr))
14331                                 continue;

14333                         if (new_ipif->ipif_flags & IPIF_POINTOPOINT)
14334                                 new_ipif->ipif_flags |= IPIF_UNNUMBERED;
14335                         else if (ipif->ipif_flags & IPIF_POINTOPOINT)
14336                                 ipif->ipif_flags |= IPIF_UNNUMBERED;
14337                         else if ((IN6_IS_ADDR_LINKLOCAL(&our_v6addr) ||
14338                             IN6_IS_ADDR_SITELOCAL(&our_v6addr)) &&
14339                             !IS_ON_SAME_LAN(ill, new_ipif->ipif_ill))
14340                                 continue;
14341                         else if (new_ipif->ipif_zoneid != ipif->ipif_zoneid &&
14342                             ipif->ipif_zoneid != ALL_ZONES && IS_LOOPBACK(ill))
14343                                 continue;
14344                         else if (new_ipif->ipif_ill == ill)
14345                                 return (EADDRINUSE);
14346                         else
14347                                 return (EADDRNOTAVAIL);
14348                 }
14349         }

14351         return (0);
14352 }

14354 /*
14355  * Bring up an ipif: bring up arp/ndp, bring up the DLPI stream, and add
14356  * IREs for the ipif.
14357  * When the routine returns EINPROGRESS then mp has been consumed and
14358  * the ioctl will be acked from ip_rput_dlpi.
14359  */
14360 int
14361 ipif_up(ipif_t *ipif, queue_t *q, mblk_t *mp)
14362 {
14363         ill_t           *ill = ipif->ipif_ill;
14364         boolean_t       isv6 = ipif->ipif_isv6;
14365         int             err = 0;
14366         boolean_t       success;
14367         uint_t          ipif_orig_id;
14368         ip_stack_t      *ipst = ill->ill_ipst;

14370         ASSERT(IAM_WRITER_IPIF(ipif));

14372         ip1dbg(("ipif_up(%s:%u)\n", ill->ill_name, ipif->ipif_id));
14373         DTRACE_PROBE3(ipif__downup, char *, "ipif_up",
14374             ill_t *, ill, ipif_t *, ipif);

14376         /* Shouldn't get here if it is already up. */
14377         if (ipif->ipif_flags & IPIF_UP)
14378                 return (EALREADY);

14380         /*
14381          * If this is a request to bring up a data address on an interface
14382          * under IPMP, then move the address to its IPMP meta-interface and
14383          * try to bring it up.  One complication is that the zeroth ipif for
```

```
14384          * an ill is special, in that every ill always has one, and that code
14385          * throughout IP deferences ill->ill_ipif without holding any locks.
14386          */
14387         if (IS_UNDER_IPMP(ill) && ipmp_ipif_is_dataaddr(ipif) &&
14388             (!ipif->ipif_isv6 || !V6_IPIF_LINKLOCAL(ipif))) {
14389                 ipif_t  *stubipif = NULL, *moveipif = NULL;
14390                 ill_t   *ipmp_ill = ipmp_illgrp_ipmp_ill(ill->ill_grp);
14391
14392                 /*
14393                  * The ipif being brought up should be quiesced.  If it's not,
14394                  * something has gone amiss and we need to bail out.  (If it's
14395                  * quiesced, we know it will remain so via IPIF_CONDEMNED.)
14396                  */
14397                 mutex_enter(&ill->ill_lock);
14398                 if (!ipif_is_quiescent(ipif)) {
14399                         mutex_exit(&ill->ill_lock);
14400                         return (EINVAL);
14401                 }
14402                 mutex_exit(&ill->ill_lock);
14403
14404                 /*
14405                  * If we're going to need to allocate ipifs, do it prior
14406                  * to starting the move (and grabbing locks).
14407                  */
14408                 if (ipif->ipif_id == 0) {
14409                         if ((moveipif = ipif_allocate(ill, 0, IRE_LOCAL, B_TRUE,
14410                             B_FALSE, &err)) == NULL) {
14411                                 return (err);
14412                         }
14413                         if ((stubipif = ipif_allocate(ill, 0, IRE_LOCAL, B_TRUE,
14414                             B_FALSE, &err)) == NULL) {
14415                                 mi_free(moveipif);
14416                                 return (err);
14417                         }
14418                 }
14419
14420                 /*
14421                  * Grab or transfer the ipif to move.  During the move, keep
14422                  * ill_g_lock held to prevent any ill walker threads from
14423                  * seeing things in an inconsistent state.
14424                  */
14425                 rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
14426                 if (ipif->ipif_id != 0) {
14427                         ipif_remove(ipif);
14428                 } else {
14429                         ipif_transfer(ipif, moveipif, stubipif);
14430                         ipif = moveipif;
14431                 }
14432
14433                 /*
14434                  * Place the ipif on the IPMP ill.  If the zeroth ipif on
14435                  * the IPMP ill is a stub (0.0.0.0 down address) then we
14436                  * replace that one.  Otherwise, pick the next available slot.
14437                  */
14438                 ipif->ipif_ill = ipmp_ill;
14439                 ipif_orig_id = ipif->ipif_id;
14440
14441                 if (ipmp_ipif_is_stubaddr(ipmp_ill->ill_ipif)) {
14442                         ipif_transfer(ipif, ipmp_ill->ill_ipif, NULL);
14443                         ipif = ipmp_ill->ill_ipif;
14444                 } else {
14445                         ipif->ipif_id = -1;
14446                         if ((err = ipif_insert(ipif, B_FALSE)) != 0) {
14447                                 /*
14448                                  * No more available ipif_id's -- put it back
14449                                  * on the original ill and fail the operation.
```



```
14450                                  * Since we're writer on the ill, we can be
14451                                  * sure our old slot is still available.
14452                                  */
14453                                 ipif->ipif_id = ipif_orig_id;
14454                                 ipif->ipif_ill = ill;
14455                                 if (ipif_orig_id == 0) {
14456                                         ipif_transfer(ipif, ill->ill_ipif,
14457                                             NULL);
14458                                 } else {
14459                                         VERIFY(ipif_insert(ipif, B_FALSE) == 0);
14460                                 }
14461                                 rw_exit(&ipst->ips_ill_g_lock);
14462                                 return (err);
14463                         }
14464                 }
14465                 rw_exit(&ipst->ips_ill_g_lock);
14466
14467                 /*
14468                  * Tell SCTP that the ipif has moved.  Note that even if we
14469                  * had to allocate a new ipif, the original sequence id was
14470                  * preserved and therefore SCTP won't know.
14471                  */
14472                 sctp_move_ipif(ipif, ill, ipmp_ill);
14473
14474                 /*
14475                  * If the ipif being brought up was on slot zero, then we
14476                  * first need to bring up the placeholder we stuck there.  In
14477                  * ip_rput_dlpi_writer(), arp_bringup_done(), or the recursive
14478                  * call to ipif_up() itself, if we successfully bring up the
14479                  * placeholder, we'll check ill_move_ipif and bring it up too.
14480                  */
14481                 if (ipif_orig_id == 0) {
14482                         ASSERT(ill->ill_move_ipif == NULL);
14483                         ill->ill_move_ipif = ipif;
14484                         if ((err = ipif_up(ill->ill_ipif, q, mp)) == 0)
14485                                 ASSERT(ill->ill_move_ipif == NULL);
14486                         if (err != EINPROGRESS)
14487                                 ill->ill_move_ipif = NULL;
14488                         return (err);
14489                 }
14490
14491                 /*
14492                  * Bring it up on the IPMP ill.
14493                  */
14494                 return (ipif_up(ipif, q, mp));
14495         }
14496
14497         /* Skip arp/ndp for any loopback interface. */
14498         if (ill->ill_wq != NULL) {
14499                 conn_t *connp = CONN_Q(q) ? Q_TO_CONN(q) : NULL;
14500                 ipsq_t  *ipsq = ill->ill_phyint->phyint_ipsq;
14501
14502                 if (!ill->ill_dl_up) {
14503                         /*
14504                          * ill_dl_up is not yet set. i.e. we are yet to
14505                          * DL_BIND with the driver and this is the first
14506                          * logical interface on the ill to become "up".
14507                          * Tell the driver to get going (via DL_BIND_REQ).
14508                          * Note that changing "significant" IFF_ flags
14509                          * address/netmask etc cause a down/up dance, but
14510                          * does not cause an unbind (DL_UNBIND) with the driver
14511                          */
14512                         return (ill_dl_up(ill, ipif, mp, q));
14513                 }
14514
14515                 /*
```

```
14516                  * ipif_resolver_up may end up needeing to bind/attach
14517                  * the ARP stream, which in turn necessitates a
14518                  * DLPI message exchange with the driver. ioctls are
14519                  * serialized and so we cannot send more than one
14520                  * interface up message at a time. If ipif_resolver_up
14521                  * does need to wait for the DLPI handshake for the ARP stream,
14522                  * we get EINPROGRESS and we will complete in arp_bringup_done.
14523                  */

14525                 ASSERT(connp != NULL || !CONN_Q(q));
14526                 if (connp != NULL)
14527                         mutex_enter(&connp->conn_lock);
14528                 mutex_enter(&ill->ill_lock);
14529                 success = ipsq_pending_mp_add(connp, ipif, q, mp, 0);
14530                 mutex_exit(&ill->ill_lock);
14531                 if (connp != NULL)
14532                         mutex_exit(&connp->conn_lock);
14533                 if (!success)
14534                         return (EINTR);

14536                 /*
14537                  * Crank up IPv6 neighbor discovery. Unlike ARP, this should
14538                  * complete when ipif_ndp_up returns.
14539                  */
14540                 err = ipif_resolver_up(ipif, Res_act_initial);
14541                 if (err == EINPROGRESS) {
14542                         /* We will complete it in arp_bringup_done() */
14543                         return (err);
14544                 }

14546                 if (isv6 && err == 0)
14547                         err = ipif_ndp_up(ipif, B_TRUE);

14549                 ASSERT(err != EINPROGRESS);
14550                 mp = ipsq_pending_mp_get(ipsq, &connp);
14551                 ASSERT(mp != NULL);
14552                 if (err != 0)
14553                         return (err);
14554         } else {
14555                 /*
14556                  * Interfaces without underlying hardware don't do duplicate
14557                  * address detection.
14558                  */
14559                 ASSERT(!(ipif->ipif_flags & IPIF_DUPLICATE));
14560                 ipif->ipif_addr_ready = 1;
14561                 err = ill_add_ires(ill);
14562                 /* allocation failure? */
14563                 if (err != 0)
14564                         return (err);
14565         }

14567         err = (isv6 ? ipif_up_done_v6(ipif) : ipif_up_done(ipif));
14568         if (err == 0 && ill->ill_move_ipif != NULL) {
14569                 ipif = ill->ill_move_ipif;
14570                 ill->ill_move_ipif = NULL;
14571                 return (ipif_up(ipif, q, mp));
14572         }
14573         return (err);
14574 }

14576 /*
14577  * Add any IREs tied to the ill. For now this is just an IRE_MULTICAST.
14578  * The identical set of IREs need to be removed in ill_delete_ires().
14579  */
14580 int
14581 ill_add_ires(ill_t *ill)
```



```
14582 {
14583         ire_t   *ire;
14584         in6_addr_t dummy6 = {(uint32_t)V6_MCAST, 0, 0, 1};
14585         in_addr_t dummy4 = htonl(INADDR_ALLHOSTS_GROUP);

14587         if (ill->ill_ire_multicast != NULL)
14588                 return (0);

14590         /*
14591          * provide some dummy ire_addr for creating the ire.
14592          */
14593         if (ill->ill_isv6) {
14594                 ire = ire_create_v6(&dummy6, 0, 0, IRE_MULTICAST, ill,
14595                     ALL_ZONES, RTF_UP, NULL, ill->ill_ipst);
14596         } else {
14597                 ire = ire_create((uchar_t *)&dummy4, 0, 0, IRE_MULTICAST, ill,
14598                     ALL_ZONES, RTF_UP, NULL, ill->ill_ipst);
14599         }
14600         if (ire == NULL)
14601                 return (ENOMEM);

14603         ill->ill_ire_multicast = ire;
14604         return (0);
14605 }

14607 void
14608 ill_delete_ires(ill_t *ill)
14609 {
14610         if (ill->ill_ire_multicast != NULL) {
14611                 /*
14612                  * BIND/ATTACH completed; Release the ref for ill_ire_multicast
14613                  * which was taken without any th_tracing enabled.
14614                  * We also mark it as condemned (note that it was never added)
14615                  * so that caching conn's can move off of it.
14616                  */
14617                 ire_make_condemned(ill->ill_ire_multicast);
14618                 ire_refrele_notr(ill->ill_ire_multicast);
14619                 ill->ill_ire_multicast = NULL;
14620         }
14621 }

14623 /*
14624  * Perform a bind for the physical device.
14625  * When the routine returns EINPROGRESS then mp has been consumed and
14626  * the ioctl will be acked from ip_rput_dlpi.
14627  * Allocate an unbind message and save it until ipif_down.
14628  */
14629 static int
14630 ill_dl_up(ill_t *ill, ipif_t *ipif, mblk_t *mp, queue_t *q)
14631 {
14632         mblk_t  *bind_mp = NULL;
14633         mblk_t  *unbind_mp = NULL;
14634         conn_t  *connp;
14635         boolean_t success;
14636         int     err;

14638         DTRACE_PROBE2(ill__downup, char *, "ill_dl_up", ill_t *, ill);

14640         ip1dbg(("ill_dl_up(%s)\n", ill->ill_name));
14641         ASSERT(IAM_WRITER_ILL(ill));
14642         ASSERT(mp != NULL);

14644         /*
14645          * Make sure we have an IRE_MULTICAST in case we immediately
14646          * start receiving packets.
14647          */
```

```
14648         err = ill_add_ires(ill);
14649         if (err != 0)
14650                 goto bad;

14652         bind_mp = ip_dlpi_alloc(sizeof (dl_bind_req_t) + sizeof (long),
14653             DL_BIND_REQ);
14654         if (bind_mp == NULL)
14655                 goto bad;
14656         ((dl_bind_req_t *)bind_mp->b_rptr)->dl_sap = ill->ill_sap;
14657         ((dl_bind_req_t *)bind_mp->b_rptr)->dl_service_mode = DL_CLDLS;

14659         /*
14660          * ill_unbind_mp would be non-null if the following sequence had
14661          * happened:
14662          * - send DL_BIND_REQ to driver, wait for response
14663          * - multiple ioctls that need to bring the ipif up are encountered,
14664          *   but they cannot enter the ipsq due to the outstanding DL_BIND_REQ.
14665          *   These ioctls will then be enqueued on the ipsq
14666          * - a DL_ERROR_ACK is returned for the DL_BIND_REQ
14667          * At this point, the pending ioctls in the ipsq will be drained, and
14668          * since ill->ill_dl_up was not set, ill_dl_up would be invoked with
14669          * a non-null ill->ill_unbind_mp
14670          */
14671         if (ill->ill_unbind_mp == NULL) {
14672                 unbind_mp = ip_dlpi_alloc(sizeof (dl_unbind_req_t),
14673                     DL_UNBIND_REQ);
14674                 if (unbind_mp == NULL)
14675                         goto bad;
14676         }
14677         /*
14678          * Record state needed to complete this operation when the
14679          * DL_BIND_ACK shows up.  Also remember the pre-allocated mblks.
14680          */
14681         connp = CONN_Q(q) ? Q_TO_CONN(q) : NULL;
14682         ASSERT(connp != NULL || !CONN_Q(q));
14683         GRAB_CONN_LOCK(q);
14684         mutex_enter(&ipif->ipif_ill->ill_lock);
14685         success = ipsq_pending_mp_add(connp, ipif, q, mp, 0);
14686         mutex_exit(&ipif->ipif_ill->ill_lock);
14687         RELEASE_CONN_LOCK(q);
14688         if (!success)
14689                 goto bad;

14691         /*
14692          * Save the unbind message for ill_dl_down(); it will be consumed when
14693          * the interface goes down.
14694          */
14695         if (ill->ill_unbind_mp == NULL)
14696                 ill->ill_unbind_mp = unbind_mp;

14698         ill_dlpi_send(ill, bind_mp);
14699         /* Send down link-layer capabilities probe if not already done. */
14700         ill_capability_probe(ill);

14702         /*
14703          * Sysid used to rely on the fact that netboots set domainname
14704          * and the like. Now that miniroot boots aren't strictly netboots
14705          * and miniroot network configuration is driven from userland
14706          * these things still need to be set. This situation can be detected
14707          * by comparing the interface being configured here to the one
14708          * dhcifname was set to reference by the boot loader. Once sysid is
14709          * converted to use dhcp_ipc_getinfo() this call can go away.
14710          */
14711         if ((ipif->ipif_flags & IPIF_DHCPRUNNING) &&
14712             (strcmp(ill->ill_name, dhcifname) == 0) &&
14713             (strlen(srpc_domain) == 0)) {
```



```
14714                 if (dhcpinit() != 0)
14715                         cmn_err(CE_WARN, "no cached dhcp response");
14716         }

14718         /*
14719          * This operation will complete in ip_rput_dlpi with either
14720          * a DL_BIND_ACK or DL_ERROR_ACK.
14721          */
14722         return (EINPROGRESS);
14723 bad:
14724         ip1dbg(("ill_dl_up(%s) FAILED\n", ill->ill_name));

14726         freemsg(bind_mp);
14727         freemsg(unbind_mp);
14728         return (ENOMEM);
14729 }

14731 /* Add room for tcp+ip headers */
14732 uint_t ip_loopback_mtuplus = IP_LOOPBACK_MTU + IP_SIMPLE_HDR_LENGTH + 20;

14734 /*
14735  * DLPI and ARP is up.
14736  * Create all the IREs associated with an interface. Bring up multicast.
14737  * Set the interface flag and finish other initialization
14738  * that potentially had to be deferred to after DL_BIND_ACK.
14739  */
14740 int
14741 ipif_up_done(ipif_t *ipif)
14742 {
14743         ill_t           *ill = ipif->ipif_ill;
14744         int             err = 0;
14745         boolean_t       loopback = B_FALSE;
14746         boolean_t       update_src_selection = B_TRUE;
14747         ipif_t          *tmp_ipif;

14749         ip1dbg(("ipif_up_done(%s:%u)\n",
14750             ipif->ipif_ill->ill_name, ipif->ipif_id));
14751         DTRACE_PROBE3(ipif__downup, char *, "ipif_up_done",
14752             ill_t *, ill, ipif_t *, ipif);

14754         /* Check if this is a loopback interface */
14755         if (ipif->ipif_ill->ill_wq == NULL)
14756                 loopback = B_TRUE;

14758         ASSERT(!MUTEX_HELD(&ipif->ipif_ill->ill_lock));

14760         /*
14761          * If all other interfaces for this ill are down or DEPRECATED,
14762          * or otherwise unsuitable for source address selection,
14763          * reset the src generation numbers to make sure source
14764          * address selection gets to take this new ipif into account.
14765          * No need to hold ill_lock while traversing the ipif list since
14766          * we are writer
14767          */
14768         for (tmp_ipif = ill->ill_ipif; tmp_ipif;
14769             tmp_ipif = tmp_ipif->ipif_next) {
14770                 if (((tmp_ipif->ipif_flags &
14771                     (IPIF_NOXMIT|IPIF_ANYCAST|IPIF_NOLOCAL|IPIF_DEPRECATED)) ||
14772                     !(tmp_ipif->ipif_flags & IPIF_UP)) ||
14773                     (tmp_ipif == ipif))
14774                         continue;
14775                 /* first useable pre-existing interface */
14776                 update_src_selection = B_FALSE;
14777                 break;
14778         }
14779         if (update_src_selection)
```

```
14780                 ip_update_source_selection(ill->ill_ipst);

14782         if (IS_LOOPBACK(ill) || ill->ill_net_type == IRE_IF_NORESOLVER) {
14783                 nce_t *loop_nce = NULL;
14784                 uint16_t flags = (NCE_F_MYADDR | NCE_F_AUTHORITY | NCE_F_NONUD);

14786                 /*
14787                  * lo0:1 and subsequent ipifs were marked IRE_LOCAL in
14788                  * ipif_lookup_on_name(), but in the case of zones we can have
14789                  * several loopback addresses on lo0. So all the interfaces with
14790                  * loopback addresses need to be marked IRE_LOOPBACK.
14791                  */
14792                 if (V4_PART_OF_V6(ipif->ipif_v6lcl_addr) ==
14793                     htonl(INADDR_LOOPBACK))
14794                         ipif->ipif_ire_type = IRE_LOOPBACK;
14795                 else
14796                         ipif->ipif_ire_type = IRE_LOCAL;
14797                 if (ill->ill_net_type != IRE_LOOPBACK)
14798                         flags |= NCE_F_PUBLISH;

14800                 /* add unicast nce for the local addr */
14801                 err = nce_lookup_then_add_v4(ill, NULL,
14802                     ill->ill_phys_addr_length, &ipif->ipif_lcl_addr, flags,
14803                     ND_REACHABLE, &loop_nce);
14804                 /* A shared-IP zone sees EEXIST for lo0:N */
14805                 if (err == 0 || err == EEXIST) {
14806                         ipif->ipif_added_nce = 1;
14807                         loop_nce->nce_ipif_cnt++;
14808                         nce_refrele(loop_nce);
14809                         err = 0;
14810                 } else {
14811                         ASSERT(loop_nce == NULL);
14812                         return (err);
14813                 }
14814         }

14816         /* Create all the IREs associated with this interface */
14817         err = ipif_add_ires_v4(ipif, loopback);
14818         if (err != 0) {
14819                 /*
14820                  * see comments about return value from
14821                  * ip_addr_availability_check() in ipif_add_ires_v4().
14822                  */
14823                 if (err != EADDRINUSE) {
14824                         (void) ipif_arp_down(ipif);
14825                 } else {
14826                         /*
14827                          * Make IPMP aware of the deleted ipif so that
14828                          * the needed ipmp cleanup (e.g., of ipif_bound_ill)
14829                          * can be completed. Note that we do not want to
14830                          * destroy the nce that was created on the ipmp_ill
14831                          * for the active copy of the duplicate address in
14832                          * use.
14833                          */
14834                         if (IS_IPMP(ill))
14835                                 ipmp_illgrp_del_ipif(ill->ill_grp, ipif);
14836                         err = EADDRNOTAVAIL;
14837                 }
14838                 return (err);
14839         }

14841         if (ill->ill_ipif_up_count == 1 && !loopback) {
14842                 /* Recover any additional IREs entries for this ill */
14843                 (void) ill_recover_saved_ire(ill);
14844         }
```



```
14846         if (ill->ill_need_recover_multicast) {
14847                 /*
14848                  * Need to recover all multicast memberships in the driver.
14849                  * This had to be deferred until we had attached.  The same
14850                  * code exists in ipif_up_done_v6() to recover IPv6
14851                  * memberships.
14852                  *
14853                  * Note that it would be preferable to unconditionally do the
14854                  * ill_recover_multicast() in ill_dl_up(), but we cannot do
14855                  * that since ill_join_allmulti() depends on ill_dl_up being
14856                  * set, and it is not set until we receive a DL_BIND_ACK after
14857                  * having called ill_dl_up().
14858                  */
14859                 ill_recover_multicast(ill);
14860         }

14862         if (ill->ill_ipif_up_count == 1) {
14863                 /*
14864                  * Since the interface is now up, it may now be active.
14865                  */
14866                 if (IS_UNDER_IPMP(ill))
14867                         ipmp_ill_refresh_active(ill);

14869                 /*
14870                  * If this is an IPMP interface, we may now be able to
14871                  * establish ARP entries.
14872                  */
14873                 if (IS_IPMP(ill))
14874                         ipmp_illgrp_refresh_arpent(ill->ill_grp);
14875         }

14877         /* Join the allhosts multicast address */
14878         ipif_multicast_up(ipif);

14880         if (!loopback && !update_src_selection &&
14881             !(ipif->ipif_flags & (IPIF_NOLOCAL|IPIF_ANYCAST|IPIF_DEPRECATED)))
14882                 ip_update_source_selection(ill->ill_ipst);

14884         if (!loopback && ipif->ipif_addr_ready) {
14885                 /* Broadcast an address mask reply. */
14886                 ipif_mask_reply(ipif);
14887         }
14888         /* Perhaps ilgs should use this ill */
14889         update_conn_ill(NULL, ill->ill_ipst);

14891         /*
14892          * This had to be deferred until we had bound.  Tell routing sockets and
14893          * others that this interface is up if it looks like the address has
14894          * been validated.  Otherwise, if it isn't ready yet, wait for
14895          * duplicate address detection to do its thing.
14896          */
14897         if (ipif->ipif_addr_ready)
14898                 ipif_up_notify(ipif);
14899         return (0);
14900 }

14902 /*
14903  * Add the IREs associated with the ipif.
14904  * Those MUST be explicitly removed in ipif_delete_ires_v4.
14905  */
14906 static int
14907 ipif_add_ires_v4(ipif_t *ipif, boolean_t loopback)
14908 {
14909         ill_t           *ill = ipif->ipif_ill;
14910         ip_stack_t      *ipst = ill->ill_ipst;
14911         ire_t           *ire_array[20];
```

```
14912         ire_t           **irep = ire_array;
14913         ire_t           **irep1;
14914         ipaddr_t        net_mask = 0;
14915         ipaddr_t        subnet_mask, route_mask;
14916         int             err;
14917         ire_t           *ire_local = NULL;      /* LOCAL or LOOPBACK */
14918         ire_t           *ire_if = NULL;
14919         uchar_t         *gw;

14921         if ((ipif->ipif_lcl_addr != INADDR_ANY) &&
14922             !(ipif->ipif_flags & IPIF_NOLOCAL)) {
14923                 /*
14924                  * If we're on a labeled system then make sure that zone-
14925                  * private addresses have proper remote host database entries.
14926                  */
14927                 if (is_system_labeled() &&
14928                     ipif->ipif_ire_type != IRE_LOOPBACK &&
14929                     !tsol_check_interface_address(ipif))
14930                         return (EINVAL);

14932                 /* Register the source address for __sin6_src_id */
14933                 err = ip_srcid_insert(&ipif->ipif_v6lcl_addr,
14934                     ipif->ipif_zoneid, ipst);
14935                 if (err != 0) {
14936                         ip0dbg(("ipif_add_ires: srcid_insert %d\n", err));
14937                         return (err);
14938                 }

14940                 if (loopback)
14941                         gw = (uchar_t *)&ipif->ipif_lcl_addr;
14942                 else
14943                         gw = NULL;

14945                 /* If the interface address is set, create the local IRE. */
14946                 ire_local = ire_create(
14947                     (uchar_t *)&ipif->ipif_lcl_addr,    /* dest address */
14948                     (uchar_t *)&ip_g_all_ones,          /* mask */
14949                     gw,                                 /* gateway */
14950                     ipif->ipif_ire_type,                /* LOCAL or LOOPBACK */
14951                     ipif->ipif_ill,
14952                     ipif->ipif_zoneid,
14953                     ((ipif->ipif_flags & IPIF_PRIVATE) ?
14954                     RTF_PRIVATE : 0) | RTF_KERNEL,
14955                     NULL,
14956                     ipst);
14957                 ip1dbg(("ipif_add_ires: 0x%p creating IRE %p type 0x%x"
14958                     " for 0x%x\n", (void *)ipif, (void *)ire_local,
14959                     ipif->ipif_ire_type,
14960                     ntohl(ipif->ipif_lcl_addr)));
14961                 if (ire_local == NULL) {
14962                         ip1dbg(("ipif_up_done: NULL ire_local\n"));
14963                         err = ENOMEM;
14964                         goto bad;
14965                 }
14966         } else {
14967                 ip1dbg((
14968                     "ipif_add_ires: not creating IRE %d for 0x%x: flags 0x%x\n",
14969                     ipif->ipif_ire_type,
14970                     ntohl(ipif->ipif_lcl_addr),
14971                     (uint_t)ipif->ipif_flags));
14972         }
14973         if ((ipif->ipif_lcl_addr != INADDR_ANY) &&
14974             !(ipif->ipif_flags & IPIF_NOLOCAL)) {
14975                 net_mask = ip_net_mask(ipif->ipif_lcl_addr);
14976         } else {
14977                 net_mask = htonl(IN_CLASSA_NET);        /* fallback */
```



```
14978         }

14980         subnet_mask = ipif->ipif_net_mask;

14982         /*
14983          * If mask was not specified, use natural netmask of
14984          * interface address. Also, store this mask back into the
14985          * ipif struct.
14986          */
14987         if (subnet_mask == 0) {
14988                 subnet_mask = net_mask;
14989                 V4MASK_TO_V6(subnet_mask, ipif->ipif_v6net_mask);
14990                 V6_MASK_COPY(ipif->ipif_v6lcl_addr, ipif->ipif_v6net_mask,
14991                     ipif->ipif_v6subnet);
14992         }

14994         /* Set up the IRE_IF_RESOLVER or IRE_IF_NORESOLVER, as appropriate. */
14995         if (!loopback && !(ipif->ipif_flags & IPIF_NOXMIT) &&
14996             ipif->ipif_subnet != INADDR_ANY) {
14997                 /* ipif_subnet is ipif_pp_dst_addr for pt-pt */

14999                 if (ipif->ipif_flags & IPIF_POINTOPOINT) {
15000                         route_mask = IP_HOST_MASK;
15001                 } else {
15002                         route_mask = subnet_mask;
15003                 }

15005                 ip1dbg(("ipif_add_ires: ipif 0x%p ill 0x%p "
15006                     "creating if IRE ill_net_type 0x%x for 0x%x\n",
15007                     (void *)ipif, (void *)ill, ill->ill_net_type,
15008                     ntohl(ipif->ipif_subnet)));
15009                 ire_if = ire_create(
15010                     (uchar_t *)&ipif->ipif_subnet,
15011                     (uchar_t *)&route_mask,
15012                     (uchar_t *)&ipif->ipif_lcl_addr,
15013                     ill->ill_net_type,
15014                     ill,
15015                     ipif->ipif_zoneid,
15016                     ((ipif->ipif_flags & IPIF_PRIVATE) ?
15017                     RTF_PRIVATE: 0) | RTF_KERNEL,
15018                     NULL,
15019                     ipst);
15020                 if (ire_if == NULL) {
15021                         ip1dbg(("ipif_up_done: NULL ire_if\n"));
15022                         err = ENOMEM;
15023                         goto bad;
15024                 }
15025         }

15027         /*
15028          * Create any necessary broadcast IREs.
15029          */
15030         if ((ipif->ipif_flags & IPIF_BROADCAST) &&
15031             !(ipif->ipif_flags & IPIF_NOXMIT))
15032                 irep = ipif_create_bcast_ires(ipif, irep);

15034         /* If an earlier ire_create failed, get out now */
15035         for (irep1 = irep; irep1 > ire_array; ) {
15036                 irep1--;
15037                 if (*irep1 == NULL) {
15038                         ip1dbg(("ipif_up_done: NULL ire found in ire_array\n"));
15039                         err = ENOMEM;
15040                         goto bad;
15041                 }
15042         }
```

```
15044         /*
15045          * Need to atomically check for IP address availability under
15046          * ip_addr_avail_lock.  ill_g_lock is held as reader to ensure no new
15047          * ills or new ipifs can be added while we are checking availability.
15048          */
15049         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
15050         mutex_enter(&ipst->ips_ip_addr_avail_lock);
15051         /* Mark it up, and increment counters. */
15052         ipif->ipif_flags |= IPIF_UP;
15053         ill->ill_ipif_up_count++;
15054         err = ip_addr_availability_check(ipif);
15055         mutex_exit(&ipst->ips_ip_addr_avail_lock);
15056         rw_exit(&ipst->ips_ill_g_lock);

15058         if (err != 0) {
15059                 /*
15060                  * Our address may already be up on the same ill. In this case,
15061                  * the ARP entry for our ipif replaced the one for the other
15062                  * ipif. So we don't want to delete it (otherwise the other ipif
15063                  * would be unable to send packets).
15064                  * ip_addr_availability_check() identifies this case for us and
15065                  * returns EADDRINUSE; Caller should turn it into EADDRNOTAVAIL
15066                  * which is the expected error code.
15067                  */
15068                 ill->ill_ipif_up_count--;
15069                 ipif->ipif_flags &= ~IPIF_UP;
15070                 goto bad;
15071         }

15073         /*
15074          * Add in all newly created IREs.  ire_create_bcast() has
15075          * already checked for duplicates of the IRE_BROADCAST type.
15076          * We add the IRE_INTERFACE before the IRE_LOCAL to ensure
15077          * that lookups find the IRE_LOCAL even if the IRE_INTERFACE is
15078          * a /32 route.
15079          */
15080         if (ire_if != NULL) {
15081                 ire_if = ire_add(ire_if);
15082                 if (ire_if == NULL) {
15083                         err = ENOMEM;
15084                         goto bad2;
15085                 }
15086 #ifdef DEBUG
15087                 ire_refhold_notr(ire_if);
15088                 ire_refrele(ire_if);
15089 #endif
15090         }
15091         if (ire_local != NULL) {
15092                 ire_local = ire_add(ire_local);
15093                 if (ire_local == NULL) {
15094                         err = ENOMEM;
15095                         goto bad2;
15096                 }
15097 #ifdef DEBUG
15098                 ire_refhold_notr(ire_local);
15099                 ire_refrele(ire_local);
15100 #endif
15101         }
15102         rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
15103         if (ire_local != NULL)
15104                 ipif->ipif_ire_local = ire_local;
15105         if (ire_if != NULL)
15106                 ipif->ipif_ire_if = ire_if;
15107         rw_exit(&ipst->ips_ill_g_lock);
15108         ire_local = NULL;
15109         ire_if = NULL;
```



```
15111         /*
15112          * We first add all of them, and if that succeeds we refrele the
15113          * bunch. That enables us to delete all of them should any of the
15114          * ire_adds fail.
15115          */
15116         for (irep1 = irep; irep1 > ire_array; ) {
15117                 irep1--;
15118                 ASSERT(!MUTEX_HELD(&((*irep1)->ire_ill->ill_lock)));
15119                 *irep1 = ire_add(*irep1);
15120                 if (*irep1 == NULL) {
15121                         err = ENOMEM;
15122                         goto bad2;
15123                 }
15124         }

15126         for (irep1 = irep; irep1 > ire_array; ) {
15127                 irep1--;
15128                 /* refheld by ire_add. */
15129                 if (*irep1 != NULL) {
15130                         ire_refrele(*irep1);
15131                         *irep1 = NULL;
15132                 }
15133         }

15135         if (!loopback) {
15136                 /*
15137                  * If the broadcast address has been set, make sure it makes
15138                  * sense based on the interface address.
15139                  * Only match on ill since we are sharing broadcast addresses.
15140                  */
15141                 if ((ipif->ipif_brd_addr != INADDR_ANY) &&
15142                     (ipif->ipif_flags & IPIF_BROADCAST)) {
15143                         ire_t   *ire;

15145                         ire = ire_ftable_lookup_v4(ipif->ipif_brd_addr, 0, 0,
15146                             IRE_BROADCAST, ipif->ipif_ill, ALL_ZONES, NULL,
15147                             (MATCH_IRE_TYPE | MATCH_IRE_ILL), 0, ipst, NULL);

15149                         if (ire == NULL) {
15150                                 /*
15151                                  * If there isn't a matching broadcast IRE,
15152                                  * revert to the default for this netmask.
15153                                  */
15154                                 ipif->ipif_v6brd_addr = ipv6_all_zeros;
15155                                 mutex_enter(&ipif->ipif_ill->ill_lock);
15156                                 ipif_set_default(ipif);
15157                                 mutex_exit(&ipif->ipif_ill->ill_lock);
15158                         } else {
15159                                 ire_refrele(ire);
15160                         }
15161                 }

15163         }
15164         return (0);

15166 bad2:
15167         ill->ill_ipif_up_count--;
15168         ipif->ipif_flags &= ~IPIF_UP;

15170 bad:
15171         ip1dbg(("ipif_add_ires: FAILED \n"));
15172         if (ire_local != NULL)
15173                 ire_delete(ire_local);
15174         if (ire_if != NULL)
15175                 ire_delete(ire_if);
```

```
15177         rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
15178         ire_local = ipif->ipif_ire_local;
15179         ipif->ipif_ire_local = NULL;
15180         ire_if = ipif->ipif_ire_if;
15181         ipif->ipif_ire_if = NULL;
15182         rw_exit(&ipst->ips_ill_g_lock);
15183         if (ire_local != NULL) {
15184                 ire_delete(ire_local);
15185                 ire_refrele_notr(ire_local);
15186         }
15187         if (ire_if != NULL) {
15188                 ire_delete(ire_if);
15189                 ire_refrele_notr(ire_if);
15190         }

15192         while (irep > ire_array) {
15193                 irep--;
15194                 if (*irep != NULL) {
15195                         ire_delete(*irep);
15196                 }
15197         }
15198         (void) ip_srcid_remove(&ipif->ipif_v6lcl_addr, ipif->ipif_zoneid, ipst);

15200         return (err);
15201 }

15203 /* Remove all the IREs created by ipif_add_ires_v4 */
15204 void
15205 ipif_delete_ires_v4(ipif_t *ipif)
15206 {
15207         ill_t           *ill = ipif->ipif_ill;
15208         ip_stack_t      *ipst = ill->ill_ipst;
15209         ire_t           *ire;

15211         rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
15212         ire = ipif->ipif_ire_local;
15213         ipif->ipif_ire_local = NULL;
15214         rw_exit(&ipst->ips_ill_g_lock);
15215         if (ire != NULL) {
15216                 /*
15217                  * Move count to ipif so we don't loose the count due to
15218                  * a down/up dance.
15219                  */
15220                 atomic_add_32(&ipif->ipif_ib_pkt_count, ire->ire_ib_pkt_count);

15222                 ire_delete(ire);
15223                 ire_refrele_notr(ire);
15224         }
15225         rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
15226         ire = ipif->ipif_ire_if;
15227         ipif->ipif_ire_if = NULL;
15228         rw_exit(&ipst->ips_ill_g_lock);
15229         if (ire != NULL) {
15230                 ire_delete(ire);
15231                 ire_refrele_notr(ire);
15232         }

15234         /*
15235          * Delete the broadcast IREs.
15236          */
15237         if ((ipif->ipif_flags & IPIF_BROADCAST) &&
15238             !(ipif->ipif_flags & IPIF_NOXMIT))
15239                 ipif_delete_bcast_ires(ipif);
15240 }
```



```
15242 /*
15243  * Checks for availbility of a usable source address (if there is one) when the
15244  * destination ILL has the ill_usesrc_ifindex pointing to another ILL. Note
15245  * this selection is done regardless of the destination.
15246  */
15247 boolean_t
15248 ipif_zone_avail(uint_t ifindex, boolean_t isv6, zoneid_t zoneid,
15249     ip_stack_t *ipst)
15250 {
15251         ipif_t          *ipif = NULL;
15252         ill_t           *uill;

15254         ASSERT(ifindex != 0);

15256         uill = ill_lookup_on_ifindex(ifindex, isv6, ipst);
15257         if (uill == NULL)
15258                 return (B_FALSE);

15260         mutex_enter(&uill->ill_lock);
15261         for (ipif = uill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
15262                 if (IPIF_IS_CONDEMNED(ipif))
15263                         continue;
15264                 if (ipif->ipif_flags & (IPIF_NOLOCAL|IPIF_ANYCAST))
15265                         continue;
15266                 if (!(ipif->ipif_flags & IPIF_UP))
15267                         continue;
15268                 if (ipif->ipif_zoneid != zoneid)
15269                         continue;
15270                 if (isv6 ? IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6lcl_addr) :
15271                     ipif->ipif_lcl_addr == INADDR_ANY)
15272                         continue;
15273                 mutex_exit(&uill->ill_lock);
15274                 ill_refrele(uill);
15275                 return (B_TRUE);
15276         }
15277         mutex_exit(&uill->ill_lock);
15278         ill_refrele(uill);
15279         return (B_FALSE);
15280 }

15282 /*
15283  * Find an ipif with a good local address on the ill+zoneid.
15284  */
15285 ipif_t *
15286 ipif_good_addr(ill_t *ill, zoneid_t zoneid)
15287 {
15288         ipif_t          *ipif;

15290         mutex_enter(&ill->ill_lock);
15291         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
15292                 if (IPIF_IS_CONDEMNED(ipif))
15293                         continue;
15294                 if (ipif->ipif_flags & (IPIF_NOLOCAL|IPIF_ANYCAST))
15295                         continue;
15296                 if (!(ipif->ipif_flags & IPIF_UP))
15297                         continue;
15298                 if (ipif->ipif_zoneid != zoneid &&
15299                     ipif->ipif_zoneid != ALL_ZONES && zoneid != ALL_ZONES)
15300                         continue;
15301                 if (ill->ill_isv6 ?
15302                     IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6lcl_addr) :
15303                     ipif->ipif_lcl_addr == INADDR_ANY)
15304                         continue;
15305                 ipif_refhold_locked(ipif);
15306                 mutex_exit(&ill->ill_lock);
15307                 return (ipif);
```

```
15308         }
15309         mutex_exit(&ill->ill_lock);
15310         return (NULL);
15311 }

15313 /*
15314  * IP source address type, sorted from worst to best.  For a given type,
15315  * always prefer IP addresses on the same subnet.  All-zones addresses are
15316  * suboptimal because they pose problems with unlabeled destinations.
15317  */
15318 typedef enum {
15319         IPIF_NONE,
15320         IPIF_DIFFNET_DEPRECATED,        /* deprecated and different subnet */
15321         IPIF_SAMENET_DEPRECATED,        /* deprecated and same subnet */
15322         IPIF_DIFFNET_ALLZONES,          /* allzones and different subnet */
15323         IPIF_SAMENET_ALLZONES,          /* allzones and same subnet */
15324         IPIF_DIFFNET,                   /* normal and different subnet */
15325         IPIF_SAMENET,                   /* normal and same subnet */
15326         IPIF_LOCALADDR                  /* local loopback */
15327 } ipif_type_t;

15329 /*
15330  * Pick the optimal ipif on 'ill' for sending to destination 'dst' from zone
15331  * 'zoneid'.  We rate usable ipifs from low -> high as per the ipif_type_t
15332  * enumeration, and return the highest-rated ipif.  If there's a tie, we pick
15333  * the first one, unless IPMP is used in which case we round-robin among them;
15334  * see below for more.
15335  *
15336  * Returns NULL if there is no suitable source address for the ill.
15337  * This only occurs when there is no valid source address for the ill.
15338  */
15339 ipif_t *
15340 ipif_select_source_v4(ill_t *ill, ipaddr_t dst, zoneid_t zoneid,
15341     boolean_t allow_usesrc, boolean_t *notreadyp)
15342 {
15343         ill_t   *usill = NULL;
15344         ill_t   *ipmp_ill = NULL;
15345         ipif_t  *start_ipif, *next_ipif, *ipif, *best_ipif;
15346         ipif_type_t type, best_type;
15347         tsol_tpc_t *src_rhtp, *dst_rhtp;
15348         ip_stack_t *ipst = ill->ill_ipst;
15349         boolean_t samenet;

15351         if (ill->ill_usesrc_ifindex != 0 && allow_usesrc) {
15352                 usill = ill_lookup_on_ifindex(ill->ill_usesrc_ifindex,
15353                     B_FALSE, ipst);
15354                 if (usill != NULL)
15355                         ill = usill;    /* Select source from usesrc ILL */
15356                 else
15357                         return (NULL);
15358         }

15360         /*
15361          * Test addresses should never be used for source address selection,
15362          * so if we were passed one, switch to the IPMP meta-interface.
15363          */
15364         if (IS_UNDER_IPMP(ill)) {
15365                 if ((ipmp_ill = ipmp_ill_hold_ipmp_ill(ill)) != NULL)
15366                         ill = ipmp_ill; /* Select source from IPMP ill */
15367                 else
15368                         return (NULL);
15369         }

15371         /*
15372          * If we're dealing with an unlabeled destination on a labeled system,
15373          * make sure that we ignore source addresses that are incompatible with
```



```
15374          * the destination's default label.  That destination's default label
15375          * must dominate the minimum label on the source address.
15376          */
15377         dst_rhtp = NULL;
15378         if (is_system_labeled()) {
15379                 dst_rhtp = find_tpc(&dst, IPV4_VERSION, B_FALSE);
15380                 if (dst_rhtp == NULL)
15381                         return (NULL);
15382                 if (dst_rhtp->tpc_tp.host_type != UNLABELED) {
15383                         TPC_RELE(dst_rhtp);
15384                         dst_rhtp = NULL;
15385                 }
15386         }

15388         /*
15389          * Hold the ill_g_lock as reader. This makes sure that no ipif/ill
15390          * can be deleted. But an ipif/ill can get CONDEMNED any time.
15391          * After selecting the right ipif, under ill_lock make sure ipif is
15392          * not condemned, and increment refcnt. If ipif is CONDEMNED,
15393          * we retry. Inside the loop we still need to check for CONDEMNED,
15394          * but not under a lock.
15395          */
15396         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
15397 retry:
15398         /*
15399          * For source address selection, we treat the ipif list as circular
15400          * and continue until we get back to where we started.  This allows
15401          * IPMP to vary source address selection (which improves inbound load
15402          * spreading) by caching its last ending point and starting from
15403          * there.  NOTE: we don't have to worry about ill_src_ipif changing
15404          * ills since that can't happen on the IPMP ill.
15405          */
15406         start_ipif = ill->ill_ipif;
15407         if (IS_IPMP(ill) && ill->ill_src_ipif != NULL)
15408                 start_ipif = ill->ill_src_ipif;

15410         ipif = start_ipif;
15411         best_ipif = NULL;
15412         best_type = IPIF_NONE;
15413         do {
15414                 if ((next_ipif = ipif->ipif_next) == NULL)
15415                         next_ipif = ill->ill_ipif;

15417                 if (IPIF_IS_CONDEMNED(ipif))
15418                         continue;
15419                 /* Always skip NOLOCAL and ANYCAST interfaces */
15420                 if (ipif->ipif_flags & (IPIF_NOLOCAL|IPIF_ANYCAST))
15421                         continue;
15422                 /* Always skip NOACCEPT interfaces */
15423                 if (ipif->ipif_ill->ill_flags & ILLF_NOACCEPT)
15424                         continue;
15425                 if (!(ipif->ipif_flags & IPIF_UP))
15426                         continue;

15428                 if (!ipif->ipif_addr_ready) {
15429                         if (notreadyp != NULL)
15430                                 *notreadyp = B_TRUE;
15431                         continue;
15432                 }

15434                 if (zoneid != ALL_ZONES &&
15435                     ipif->ipif_zoneid != zoneid &&
15436                     ipif->ipif_zoneid != ALL_ZONES)
15437                         continue;

15439                 /*
```

```
15440                 * Interfaces with 0.0.0.0 address are allowed to be UP, but
15441                 * are not valid as source addresses.
15442                 */
15443                if (ipif->ipif_lcl_addr == INADDR_ANY)
15444                        continue;

15446                /*
15447                 * Check compatibility of local address for destination's
15448                 * default label if we're on a labeled system.  Incompatible
15449                 * addresses can't be used at all.
15450                 */
15451                if (dst_rhtp != NULL) {
15452                        boolean_t incompat;

15454                        src_rhtp = find_tpc(&ipif->ipif_lcl_addr,
15455                            IPV4_VERSION, B_FALSE);
15456                        if (src_rhtp == NULL)
15457                                continue;
15458                        incompat = src_rhtp->tpc_tp.host_type != SUN_CIPSO ||
15459                            src_rhtp->tpc_tp.tp_doi !=
15460                            dst_rhtp->tpc_tp.tp_doi ||
15461                            (!_blinrange(&dst_rhtp->tpc_tp.tp_def_label,
15462                            &src_rhtp->tpc_tp.tp_sl_range_cipso) &&
15463                            !blinlset(&dst_rhtp->tpc_tp.tp_def_label,
15464                            src_rhtp->tpc_tp.tp_sl_set_cipso));
15465                        TPC_RELE(src_rhtp);
15466                        if (incompat)
15467                                continue;
15468                }

15470                samenet = ((ipif->ipif_net_mask & dst) == ipif->ipif_subnet);

15472                if (ipif->ipif_lcl_addr == dst) {
15473                        type = IPIF_LOCALADDR;
15474                } else if (ipif->ipif_flags & IPIF_DEPRECATED) {
15475                        type = samenet ? IPIF_SAMENET_DEPRECATED :
15476                            IPIF_DIFFNET_DEPRECATED;
15477                } else if (ipif->ipif_zoneid == ALL_ZONES) {
15478                        type = samenet ? IPIF_SAMENET_ALLZONES :
15479                            IPIF_DIFFNET_ALLZONES;
15480                } else {
15481                        type = samenet ? IPIF_SAMENET : IPIF_DIFFNET;
15482                }

15484                if (type > best_type) {
15485                        best_type = type;
15486                        best_ipif = ipif;
15487                        if (best_type == IPIF_LOCALADDR)
15488                                break; /* can't get better */
15489                }
15490        } while ((ipif = next_ipif) != start_ipif);

15492        if ((ipif = best_ipif) != NULL) {
15493                mutex_enter(&ipif->ipif_ill->ill_lock);
15494                if (IPIF_IS_CONDEMNED(ipif)) {
15495                        mutex_exit(&ipif->ipif_ill->ill_lock);
15496                        goto retry;
15497                }
15498                ipif_refhold_locked(ipif);

15500                /*
15501                 * For IPMP, update the source ipif rotor to the next ipif,
15502                 * provided we can look it up.  (We must not use it if it's
15503                 * IPIF_CONDEMNED since we may have grabbed ill_g_lock after
15504                 * ipif_free() checked ill_src_ipif.)
15505                 */
```



```
15506                if (IS_IPMP(ill) && ipif != NULL) {
15507                        next_ipif = ipif->ipif_next;
15508                        if (next_ipif != NULL && !IPIF_IS_CONDEMNED(next_ipif))
15509                                ill->ill_src_ipif = next_ipif;
15510                        else
15511                                ill->ill_src_ipif = NULL;
15512                }
15513                mutex_exit(&ipif->ipif_ill->ill_lock);
15514        }

15516        rw_exit(&ipst->ips_ill_g_lock);
15517        if (usill != NULL)
15518                ill_refrele(usill);
15519        if (ipmp_ill != NULL)
15520                ill_refrele(ipmp_ill);
15521        if (dst_rhtp != NULL)
15522                TPC_RELE(dst_rhtp);

15524 #ifdef DEBUG
15525        if (ipif == NULL) {
15526                char buf1[INET6_ADDRSTRLEN];

15528                ip1dbg(("ipif_select_source_v4(%s, %s) -> NULL\n",
15529                    ill->ill_name,
15530                    inet_ntop(AF_INET, &dst, buf1, sizeof (buf1))));
15531        } else {
15532                char buf1[INET6_ADDRSTRLEN];
15533                char buf2[INET6_ADDRSTRLEN];

15535                ip1dbg(("ipif_select_source_v4(%s, %s) -> %s\n",
15536                    ipif->ipif_ill->ill_name,
15537                    inet_ntop(AF_INET, &dst, buf1, sizeof (buf1)),
15538                    inet_ntop(AF_INET, &ipif->ipif_lcl_addr,
15539                    buf2, sizeof (buf2))));
15540        }
15541 #endif /* DEBUG */
15542        return (ipif);
15543 }

15545 /*
15546  * Pick a source address based on the destination ill and an optional setsrc
15547  * address.
15548  * The result is stored in srcp. If generation is set, then put the source
15549  * generation number there before we look for the source address (to avoid
15550  * missing changes in the set of source addresses.
15551  * If flagsp is set, then us it to pass back ipif_flags.
15552  *
15553  * If the caller wants to cache the returned source address and detect when
15554  * that might be stale, the caller should pass in a generation argument,
15555  * which the caller can later compare against ips_src_generation
15556  *
15557  * The precedence order for selecting an IPv4 source address is:
15558  *  - RTF_SETSRC on the offlink ire always wins.
15559  *  - If usrsrc is set, swap the ill to be the usesrc one.
15560  *  - If IPMP is used on the ill, select a random address from the most
15561  *    preferred ones below:
15562  * 1. If onlink destination, same subnet and not deprecated, not ALL_ZONES
15563  * 2. Not deprecated, not ALL_ZONES
15564  * 3. If onlink destination, same subnet and not deprecated, ALL_ZONES
15565  * 4. Not deprecated, ALL_ZONES
15566  * 5. If onlink destination, same subnet and deprecated
15567  * 6. Deprecated.
15568  *
15569  * We have lower preference for ALL_ZONES IP addresses,
15570  * as they pose problems with unlabeled destinations.
15571  *
```

```
15572  * Note that when multiple IP addresses match e.g., #1 we pick
15573  * the first one if IPMP is not in use. With IPMP we randomize.
15574  */
15575 int
15576 ip_select_source_v4(ill_t *ill, ipaddr_t setsrc, ipaddr_t dst,
15577     ipaddr_t multicast_ifaddr,
15578     zoneid_t zoneid, ip_stack_t *ipst, ipaddr_t *srcp,
15579     uint32_t *generation, uint64_t *flagsp)
15580 {
15581         ipif_t *ipif;
15582         boolean_t notready = B_FALSE;   /* Set if !ipif_addr_ready found */

15584         if (flagsp != NULL)
15585                 *flagsp = 0;

15587         /*
15588          * Need to grab the generation number before we check to
15589          * avoid a race with a change to the set of local addresses.
15590          * No lock needed since the thread which updates the set of local
15591          * addresses use ipif/ill locks and exit those (hence a store memory
15592          * barrier) before doing the atomic increase of ips_src_generation.
15593          */
15594         if (generation != NULL) {
15595                 *generation = ipst->ips_src_generation;
15596         }

15598         if (CLASSD(dst) && multicast_ifaddr != INADDR_ANY) {
15599                 *srcp = multicast_ifaddr;
15600                 return (0);
15601         }

15603         /* Was RTF_SETSRC set on the first IRE in the recursive lookup? */
15604         if (setsrc != INADDR_ANY) {
15605                 *srcp = setsrc;
15606                 return (0);
15607         }
15608         ipif = ipif_select_source_v4(ill, dst, zoneid, B_TRUE, &notready);
15609         if (ipif == NULL) {
15610                 if (notready)
15611                         return (ENETDOWN);
15612                 else
15613                         return (EADDRNOTAVAIL);
15614         }
15615         *srcp = ipif->ipif_lcl_addr;
15616         if (flagsp != NULL)
15617                 *flagsp = ipif->ipif_flags;
15618         ipif_refrele(ipif);
15619         return (0);
15620 }

15622 /* ARGSUSED */
15623 int
15624 if_unitsel_restart(ipif_t *ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
15625         ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
15626 {
15627         /*
15628          * ill_phyint_reinit merged the v4 and v6 into a single
15629          * ipsq.  We might not have been able to complete the
15630          * operation in ipif_set_values, if we could not become
15631          * exclusive.  If so restart it here.
15632          */
15633         return (ipif_set_values_tail(ipif->ipif_ill, ipif, mp, q));
15634 }

15636 /*
15637  * Can operate on either a module or a driver queue.
```



```
15638  * Returns an error if not a module queue.
15639  */
15640 /* ARGSUSED */
15641 int
15642 if_unitsel(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
15643     ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
15644 {
15645         queue_t         *q1 = q;
15646         char            *cp;
15647         char            interf_name[LIFNAMSIZ];
15648         uint_t          ppa = *(uint_t *)mp->b_cont->b_cont->b_rptr;

15650         if (q->q_next == NULL) {
15651                 ip1dbg((
15652                     "if_unitsel: IF_UNITSEL: no q_next\n"));
15653                 return (EINVAL);
15654         }

15656         if (((ill_t *)(q->q_ptr))->ill_name[0] != '\0')
15657                 return (EALREADY);

15659         do {
15660                 q1 = q1->q_next;
15661         } while (q1->q_next);
15662         cp = q1->q_qinfo->qi_minfo->mi_idname;
15663         (void) sprintf(interf_name, "%s%d", cp, ppa);

15665         /*
15666          * Here we are not going to delay the ioack until after
15667          * ACKs from DL_ATTACH_REQ/DL_BIND_REQ. So no need to save the
15668          * original ioctl message before sending the requests.
15669          */
15670         return (ipif_set_values(q, mp, interf_name, &ppa));
15671 }

15673 /* ARGSUSED */
15674 int
15675 ip_sioctl_sifname(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
15676     ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
15677 {
15678         return (ENXIO);
15679 }

15681 /*
15682  * Create any IRE_BROADCAST entries for 'ipif', and store those entries in
15683  * 'irep'.  Returns a pointer to the next free 'irep' entry
15684  * A mirror exists in ipif_delete_bcast_ires().
15685  *
15686  * The management of any "extra" or seemingly duplicate IRE_BROADCASTs is
15687  * done in ire_add.
15688  */
15689 static ire_t **
15690 ipif_create_bcast_ires(ipif_t *ipif, ire_t **irep)
15691 {
15692         ipaddr_t addr;
15693         ipaddr_t netmask = ip_net_mask(ipif->ipif_lcl_addr);
15694         ipaddr_t subnetmask = ipif->ipif_net_mask;
15695         ill_t *ill = ipif->ipif_ill;
15696         zoneid_t zoneid = ipif->ipif_zoneid;

15698         ip1dbg(("ipif_create_bcast_ires: creating broadcast IREs\n"));

15700         ASSERT(ipif->ipif_flags & IPIF_BROADCAST);
15701         ASSERT(!(ipif->ipif_flags & IPIF_NOXMIT));

15703         if (ipif->ipif_lcl_addr == INADDR_ANY ||
```

```
15704             (ipif->ipif_flags & IPIF_NOLOCAL))
15705                 netmask = htonl(IN_CLASSA_NET);         /* fallback */

15707         irep = ire_create_bcast(ill, 0, zoneid, irep);
15708         irep = ire_create_bcast(ill, INADDR_BROADCAST, zoneid, irep);

15710         /*
15711          * For backward compatibility, we create net broadcast IREs based on
15712          * the old "IP address class system", since some old machines only
15713          * respond to these class derived net broadcast.  However, we must not
15714          * create these net broadcast IREs if the subnetmask is shorter than
15715          * the IP address class based derived netmask.  Otherwise, we may
15716          * create a net broadcast address which is the same as an IP address
15717          * on the subnet -- and then TCP will refuse to talk to that address.
15718          */
15719         if (netmask < subnetmask) {
15720                 addr = netmask & ipif->ipif_subnet;
15721                 irep = ire_create_bcast(ill, addr, zoneid, irep);
15722                 irep = ire_create_bcast(ill, ~netmask | addr, zoneid, irep);
15723         }

15725         /*
15726          * Don't create IRE_BROADCAST IREs for the interface if the subnetmask
15727          * is 0xFFFFFFFF, as an IRE_LOCAL for that interface is already
15728          * created.  Creating these broadcast IREs will only create confusion
15729          * as 'addr' will be the same as the IP address.
15730          */
15731         if (subnetmask != 0xFFFFFFFF) {
15732                 addr = ipif->ipif_subnet;
15733                 irep = ire_create_bcast(ill, addr, zoneid, irep);
15734                 irep = ire_create_bcast(ill, ~subnetmask | addr, zoneid, irep);
15735         }

15737         return (irep);
15738 }

15740 /*
15741  * Mirror of ipif_create_bcast_ires()
15742  */
15743 static void
15744 ipif_delete_bcast_ires(ipif_t *ipif)
15745 {
15746         ipaddr_t        addr;
15747         ipaddr_t        netmask = ip_net_mask(ipif->ipif_lcl_addr);
15748         ipaddr_t        subnetmask = ipif->ipif_net_mask;
15749         ill_t           *ill = ipif->ipif_ill;
15750         zoneid_t        zoneid = ipif->ipif_zoneid;
15751         ire_t           *ire;

15753         ASSERT(ipif->ipif_flags & IPIF_BROADCAST);
15754         ASSERT(!(ipif->ipif_flags & IPIF_NOXMIT));

15756         if (ipif->ipif_lcl_addr == INADDR_ANY ||
15757             (ipif->ipif_flags & IPIF_NOLOCAL))
15758                 netmask = htonl(IN_CLASSA_NET);         /* fallback */

15760         ire = ire_lookup_bcast(ill, 0, zoneid);
15761         ASSERT(ire != NULL);
15762         ire_delete(ire); ire_refrele(ire);
15763         ire = ire_lookup_bcast(ill, INADDR_BROADCAST, zoneid);
15764         ASSERT(ire != NULL);
15765         ire_delete(ire); ire_refrele(ire);

15767         /*
15768          * For backward compatibility, we create net broadcast IREs based on
15769          * the old "IP address class system", since some old machines only
```



```
15770          * respond to these class derived net broadcast.  However, we must not
15771          * create these net broadcast IREs if the subnetmask is shorter than
15772          * the IP address class based derived netmask.  Otherwise, we may
15773          * create a net broadcast address which is the same as an IP address
15774          * on the subnet -- and then TCP will refuse to talk to that address.
15775          */
15776         if (netmask < subnetmask) {
15777                 addr = netmask & ipif->ipif_subnet;
15778                 ire = ire_lookup_bcast(ill, addr, zoneid);
15779                 ASSERT(ire != NULL);
15780                 ire_delete(ire); ire_refrele(ire);
15781                 ire = ire_lookup_bcast(ill, ~netmask | addr, zoneid);
15782                 ASSERT(ire != NULL);
15783                 ire_delete(ire); ire_refrele(ire);
15784         }

15786         /*
15787          * Don't create IRE_BROADCAST IREs for the interface if the subnetmask
15788          * is 0xFFFFFFFF, as an IRE_LOCAL for that interface is already
15789          * created.  Creating these broadcast IREs will only create confusion
15790          * as 'addr' will be the same as the IP address.
15791          */
15792         if (subnetmask != 0xFFFFFFFF) {
15793                 addr = ipif->ipif_subnet;
15794                 ire = ire_lookup_bcast(ill, addr, zoneid);
15795                 ASSERT(ire != NULL);
15796                 ire_delete(ire); ire_refrele(ire);
15797                 ire = ire_lookup_bcast(ill, ~subnetmask | addr, zoneid);
15798                 ASSERT(ire != NULL);
15799                 ire_delete(ire); ire_refrele(ire);
15800         }
15801 }

15803 /*
15804  * Extract both the flags (including IFF_CANTCHANGE) such as IFF_IPV*
15805  * from lifr_flags and the name from lifr_name.
15806  * Set IFF_IPV* and ill_isv6 prior to doing the lookup
15807  * since ipif_lookup_on_name uses the _isv6 flags when matching.
15808  * Returns EINPROGRESS when mp has been consumed by queueing it on
15809  * ipx_pending_mp and the ioctl will complete in ip_rput.
15810  *
15811  * Can operate on either a module or a driver queue.
15812  * Returns an error if not a module queue.
15813  */
15814 /* ARGSUSED */
15815 int
15816 ip_sioctl_slifname(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
15817     ip_ioctl_cmd_t *ipip, void *if_req)
15818 {
15819         ill_t   *ill = q->q_ptr;
15820         phyint_t *phyi;
15821         ip_stack_t *ipst;
15822         struct lifreq *lifr = if_req;
15823         uint64_t new_flags;

15825         ASSERT(ipif != NULL);
15826         ip1dbg(("ip_sioctl_slifname %s\n", lifr->lifr_name));

15828         if (q->q_next == NULL) {
15829                 ip1dbg(("if_sioctl_slifname: SIOCSLIFNAME: no q_next\n"));
15830                 return (EINVAL);
15831         }

15833         /*
15834          * If we are not writer on 'q' then this interface exists already
15835          * and previous lookups (ip_extract_lifreq()) found this ipif --
```

```
15836          * so return EALREADY.
15837          */
15838         if (ill != ipif->ipif_ill)
15839                 return (EALREADY);

15841         if (ill->ill_name[0] != '\0')
15842                 return (EALREADY);

15844         /*
15845          * If there's another ill already with the requested name, ensure
15846          * that it's of the same type.  Otherwise, ill_phyint_reinit() will
15847          * fuse together two unrelated ills, which will cause chaos.
15848          */
15849         ipst = ill->ill_ipst;
15850         phyi = avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
15851             lifr->lifr_name, NULL);
15852         if (phyi != NULL) {
15853                 ill_t *ill_mate = phyi->phyint_illv4;

15855                 if (ill_mate == NULL)
15856                         ill_mate = phyi->phyint_illv6;
15857                 ASSERT(ill_mate != NULL);

15859                 if (ill_mate->ill_media->ip_m_mac_type !=
15860                     ill->ill_media->ip_m_mac_type) {
15861                         ip1dbg(("if_sioctl_slifname: SIOCSLIFNAME: attempt to "
15862                             "use the same ill name on differing media\n"));
15863                         return (EINVAL);
15864                 }
15865         }

15867         /*
15868          * We start off as IFF_IPV4 in ipif_allocate and become
15869          * IFF_IPV4 or IFF_IPV6 here depending  on lifr_flags value.
15870          * The only flags that we read from user space are IFF_IPV4,
15871          * IFF_IPV6, and IFF_BROADCAST.
15872          *
15873          * This ill has not been inserted into the global list.
15874          * So we are still single threaded and don't need any lock
15875          *
15876          * Saniy check the flags.
15877          */

15879         if ((lifr->lifr_flags & IFF_BROADCAST) &&
15880             ((lifr->lifr_flags & IFF_IPV6) ||
15881             (!ill->ill_needs_attach && ill->ill_bcast_addr_length == 0))) {
15882                 ip1dbg(("ip_sioctl_slifname: link not broadcast capable "
15883                     "or IPv6 i.e., no broadcast \n"));
15884                 return (EINVAL);
15885         }

15887         new_flags =
15888             lifr->lifr_flags & (IFF_IPV6|IFF_IPV4|IFF_BROADCAST);

15890         if ((new_flags ^ (IFF_IPV6|IFF_IPV4)) == 0) {
15891                 ip1dbg(("ip_sioctl_slifname: flags must be exactly one of "
15892                     "IFF_IPV4 or IFF_IPV6\n"));
15893                 return (EINVAL);
15894         }

15896         /*
15897          * We always start off as IPv4, so only need to check for IPv6.
15898          */
15899         if ((new_flags & IFF_IPV6) != 0) {
15900                 ill->ill_flags |= ILLF_IPV6;
15901                 ill->ill_flags &= ~ILLF_IPV4;
```



```
15903                 if (lifr->lifr_flags & IFF_NOLINKLOCAL)
15904                         ill->ill_flags |= ILLF_NOLINKLOCAL;
15905         }

15907         if ((new_flags & IFF_BROADCAST) != 0)
15908                 ipif->ipif_flags |= IPIF_BROADCAST;
15909         else
15910                 ipif->ipif_flags &= ~IPIF_BROADCAST;

15912         /* We started off as V4. */
15913         if (ill->ill_flags & ILLF_IPV6) {
15914                 ill->ill_phyint->phyint_illv6 = ill;
15915                 ill->ill_phyint->phyint_illv4 = NULL;
15916         }

15918         return (ipif_set_values(q, mp, lifr->lifr_name, &lifr->lifr_ppa));
15919 }

15921 /* ARGSUSED */
15922 int
15923 ip_sioctl_slifname_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
15924     ip_ioctl_cmd_t *ipip, void *if_req)
15925 {
15926         /*
15927          * ill_phyint_reinit merged the v4 and v6 into a single
15928          * ipsq.  We might not have been able to complete the
15929          * slifname in ipif_set_values, if we could not become
15930          * exclusive.  If so restart it here
15931          */
15932         return (ipif_set_values_tail(ipif->ipif_ill, ipif, mp, q));
15933 }

15935 /*
15936  * Return a pointer to the ipif which matches the index, IP version type and
15937  * zoneid.
15938  */
15939 ipif_t *
15940 ipif_lookup_on_ifindex(uint_t index, boolean_t isv6, zoneid_t zoneid,
15941     ip_stack_t *ipst)
15942 {
15943         ill_t   *ill;
15944         ipif_t  *ipif = NULL;

15946         ill = ill_lookup_on_ifindex(index, isv6, ipst);
15947         if (ill != NULL) {
15948                 mutex_enter(&ill->ill_lock);
15949                 for (ipif = ill->ill_ipif; ipif != NULL;
15950                     ipif = ipif->ipif_next) {
15951                         if (!IPIF_IS_CONDEMNED(ipif) && (zoneid == ALL_ZONES ||
15952                             zoneid == ipif->ipif_zoneid ||
15953                             ipif->ipif_zoneid == ALL_ZONES)) {
15954                                 ipif_refhold_locked(ipif);
15955                                 break;
15956                         }
15957                 }
15958                 mutex_exit(&ill->ill_lock);
15959                 ill_refrele(ill);
15960         }
15961         return (ipif);
15962 }

15964 /*
15965  * Change an existing physical interface's index. If the new index
15966  * is acceptable we update the index and the phyint_list_avl_by_index tree.
15967  * Finally, we update other systems which may have a dependence on the
```

```
15968  * index value.
15969  */
15970 /* ARGSUSED */
15971 int
15972 ip_sioctl_slifindex(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
15973     ip_ioctl_cmd_t *ipip, void *ifreq)
15974 {
15975         ill_t           *ill;
15976         phyint_t        *phyi;
15977         struct ifreq    *ifr = (struct ifreq *)ifreq;
15978         struct lifreq   *lifr = (struct lifreq *)ifreq;
15979         uint_t  old_index, index;
15980         ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;
15981         avl_index_t     where;

15983         if (ipip->ipi_cmd_type == IF_CMD)
15984                 index = ifr->ifr_index;
15985         else
15986                 index = lifr->lifr_index;

15988         /*
15989          * Only allow on physical interface. Also, index zero is illegal.
15990          */
15991         ill = ipif->ipif_ill;
15992         phyi = ill->ill_phyint;
15993         if (ipif->ipif_id != 0 || index == 0 || index > IF_INDEX_MAX) {
15994                 return (EINVAL);
15995         }

15997         /* If the index is not changing, no work to do */
15998         if (phyi->phyint_ifindex == index)
15999                 return (0);

16001         /*
16002          * Use phyint_exists() to determine if the new interface index
16003          * is already in use. If the index is unused then we need to
16004          * change the phyint's position in the phyint_list_avl_by_index
16005          * tree. If we do not do this, subsequent lookups (using the new
16006          * index value) will not find the phyint.
16007          */
16008         rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
16009         if (phyint_exists(index, ipst)) {
16010                 rw_exit(&ipst->ips_ill_g_lock);
16011                 return (EEXIST);
16012         }

16014         /*
16015          * The new index is unused. Set it in the phyint. However we must not
16016          * forget to trigger NE_IFINDEX_CHANGE event before the ifindex
16017          * changes. The event must be bound to old ifindex value.
16018          */
16019         ill_nic_event_dispatch(ill, 0, NE_IFINDEX_CHANGE,
16020             &index, sizeof (index));

16022         old_index = phyi->phyint_ifindex;
16023         phyi->phyint_ifindex = index;

16025         avl_remove(&ipst->ips_phyint_g_list->phyint_list_avl_by_index, phyi);
16026         (void) avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
16027             &index, &where);
16028         avl_insert(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
16029             phyi, where);
16030         rw_exit(&ipst->ips_ill_g_lock);

16032         /* Update SCTP's ILL list */
16033         sctp_ill_reindex(ill, old_index);
```



```
16035         /* Send the routing sockets message */
16036         ip_rts_ifmsg(ipif, RTSQ_DEFAULT);
16037         if (ILL_OTHER(ill))
16038                 ip_rts_ifmsg(ILL_OTHER(ill)->ill_ipif, RTSQ_DEFAULT);

16040         /* Perhaps ilgs should use this ill */
16041         update_conn_ill(NULL, ill->ill_ipst);
16042         return (0);
16043 }

16045 /* ARGSUSED */
16046 int
16047 ip_sioctl_get_lifindex(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16048     ip_ioctl_cmd_t *ipip, void *ifreq)
16049 {
16050         struct ifreq    *ifr = (struct ifreq *)ifreq;
16051         struct lifreq   *lifr = (struct lifreq *)ifreq;

16053         ip1dbg(("ip_sioctl_get_lifindex(%s:%u %p)\n",
16054             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
16055         /* Get the interface index */
16056         if (ipip->ipi_cmd_type == IF_CMD) {
16057                 ifr->ifr_index = ipif->ipif_ill->ill_phyint->phyint_ifindex;
16058         } else {
16059                 lifr->lifr_index = ipif->ipif_ill->ill_phyint->phyint_ifindex;
16060         }
16061         return (0);
16062 }

16064 /* ARGSUSED */
16065 int
16066 ip_sioctl_get_lifzone(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16067     ip_ioctl_cmd_t *ipip, void *ifreq)
16068 {
16069         struct lifreq   *lifr = (struct lifreq *)ifreq;

16071         ip1dbg(("ip_sioctl_get_lifzone(%s:%u %p)\n",
16072             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
16073         /* Get the interface zone */
16074         ASSERT(ipip->ipi_cmd_type == LIF_CMD);
16075         lifr->lifr_zoneid = ipif->ipif_zoneid;
16076         return (0);
16077 }

16079 /*
16080  * Set the zoneid of an interface.
16081  */
16082 /* ARGSUSED */
16083 int
16084 ip_sioctl_slifzone(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16085     ip_ioctl_cmd_t *ipip, void *ifreq)
16086 {
16087         struct lifreq   *lifr = (struct lifreq *)ifreq;
16088         int err = 0;
16089         boolean_t need_up = B_FALSE;
16090         zone_t *zptr;
16091         zone_status_t status;
16092         zoneid_t zoneid;

16094         ASSERT(ipip->ipi_cmd_type == LIF_CMD);
16095         if ((zoneid = lifr->lifr_zoneid) == ALL_ZONES) {
16096                 if (!is_system_labeled())
16097                         return (ENOTSUP);
16098                 zoneid = GLOBAL_ZONEID;
16099         }
```

```
16101         /* cannot assign instance zero to a non-global zone */
16102         if (ipif->ipif_id == 0 && zoneid != GLOBAL_ZONEID)
16103                 return (ENOTSUP);
16105         /*
16106          * Cannot assign to a zone that doesn't exist or is shutting down.  In
16107          * the event of a race with the zone shutdown processing, since IP
16108          * serializes this ioctl and SIOCGLIFCONF/SIOCLIFREMOVEIF, we know the
16109          * interface will be cleaned up even if the zone is shut down
16110          * immediately after the status check. If the interface can't be brought
16111          * down right away, and the zone is shut down before the restart
16112          * function is called, we resolve the possible races by rechecking the
16113          * zone status in the restart function.
16114          */
16115         if ((zptr = zone_find_by_id(zoneid)) == NULL)
16116                 return (EINVAL);
16117         status = zone_status_get(zptr);
16118         zone_rele(zptr);
16120         if (status != ZONE_IS_READY && status != ZONE_IS_RUNNING)
16121                 return (EINVAL);
16123         if (ipif->ipif_flags & IPIF_UP) {
16124                 /*
16125                  * If the interface is already marked up,
16126                  * we call ipif_down which will take care
16127                  * of ditching any IREs that have been set
16128                  * up based on the old interface address.
16129                  */
16130                 err = ipif_logical_down(ipif, q, mp);
16131                 if (err == EINPROGRESS)
16132                         return (err);
16133                 (void) ipif_down_tail(ipif);
16134                 need_up = B_TRUE;
16135         }
16137         err = ip_sioctl_slifzone_tail(ipif, lifr->lifr_zoneid, q, mp, need_up);
16138         return (err);
16139 }
16141 static int
16142 ip_sioctl_slifzone_tail(ipif_t *ipif, zoneid_t zoneid,
16143     queue_t *q, mblk_t *mp, boolean_t need_up)
16144 {
16145         int     err = 0;
16146         ip_stack_t      *ipst;
16148         ip1dbg(("ip_sioctl_zoneid_tail(%s:%u %p)\n",
16149             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
16151         if (CONN_Q(q))
16152                 ipst = CONNQ_TO_IPST(q);
16153         else
16154                 ipst = ILLQ_TO_IPST(q);
16156         /*
16157          * For exclusive stacks we don't allow a different zoneid than
16158          * global.
16159          */
16160         if (ipst->ips_netstack->netstack_stackid != GLOBAL_NETSTACKID &&
16161             zoneid != GLOBAL_ZONEID)
16162                 return (EINVAL);
16164         /* Set the new zone id. */
16165         ipif->ipif_zoneid = zoneid;
```



```
16167         /* Update sctp list */
16168         sctp_update_ipif(ipif, SCTP_IPIF_UPDATE);
16170         /* The default multicast interface might have changed */
16171         ire_increment_multicast_generation(ipst, ipif->ipif_ill->ill_isv6);
16173         if (need_up) {
16174                 /*
16175                  * Now bring the interface back up.  If this
16176                  * is the only IPIF for the ILL, ipif_up
16177                  * will have to re-bind to the device, so
16178                  * we may get back EINPROGRESS, in which
16179                  * case, this IOCTL will get completed in
16180                  * ip_rput_dlpi when we see the DL_BIND_ACK.
16181                  */
16182                 err = ipif_up(ipif, q, mp);
16183         }
16184         return (err);
16185 }
16187 /* ARGSUSED */
16188 int
16189 ip_sioctl_slifzone_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16190     ip_ioctl_cmd_t *ipip, void *if_req)
16191 {
16192         struct lifreq *lifr = (struct lifreq *)if_req;
16193         zoneid_t zoneid;
16194         zone_t *zptr;
16195         zone_status_t status;
16197         ASSERT(ipip->ipi_cmd_type == LIF_CMD);
16198         if ((zoneid = lifr->lifr_zoneid) == ALL_ZONES)
16199                 zoneid = GLOBAL_ZONEID;
16201         ip1dbg(("ip_sioctl_slifzone_restart(%s:%u %p)\n",
16202             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
16204         /*
16205          * We recheck the zone status to resolve the following race condition:
16206          * 1) process sends SIOCSLIFZONE to put hme0:1 in zone "myzone";
16207          * 2) hme0:1 is up and can't be brought down right away;
16208          * ip_sioctl_slifzone() returns EINPROGRESS and the request is queued;
16209          * 3) zone "myzone" is halted; the zone status switches to
16210          * 'shutting_down' and the zones framework sends SIOCGLIFCONF to list
16211          * the interfaces to remove - hme0:1 is not returned because it's not
16212          * yet in "myzone", so it won't be removed;
16213          * 4) the restart function for SIOCSLIFZONE is called; without the
16214          * status check here, we would have hme0:1 in "myzone" after it's been
16215          * destroyed.
16216          * Note that if the status check fails, we need to bring the interface
16217          * back to its state prior to ip_sioctl_slifzone(), hence the call to
16218          * ipif_up_done[_v6]().
16219          */
16220         status = ZONE_IS_UNINITIALIZED;
16221         if ((zptr = zone_find_by_id(zoneid)) != NULL) {
16222                 status = zone_status_get(zptr);
16223                 zone_rele(zptr);
16224         }
16225         if (status != ZONE_IS_READY && status != ZONE_IS_RUNNING) {
16226                 if (ipif->ipif_isv6) {
16227                         (void) ipif_up_done_v6(ipif);
16228                 } else {
16229                         (void) ipif_up_done(ipif);
16230                 }
16231                 return (EINVAL);
```

```
16232         }

16234         (void) ipif_down_tail(ipif);

16236         return (ip_sioctl_slifzone_tail(ipif, lifr->lifr_zoneid, q, mp,
16237             B_TRUE));
16238 }

16240 /*
16241  * Return the number of addresses on 'ill' with one or more of the values
16242  * in 'set' set and all of the values in 'clear' clear.
16243  */
16244 static uint_t
16245 ill_flagaddr_cnt(const ill_t *ill, uint64_t set, uint64_t clear)
16246 {
16247         ipif_t  *ipif;
16248         uint_t  cnt = 0;

16250         ASSERT(IAM_WRITER_ILL(ill));

16252         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next)
16253                 if ((ipif->ipif_flags & set) && !(ipif->ipif_flags & clear))
16254                         cnt++;

16256         return (cnt);
16257 }

16259 /*
16260  * Return the number of migratable addresses on 'ill' that are under
16261  * application control.
16262  */
16263 uint_t
16264 ill_appaddr_cnt(const ill_t *ill)
16265 {
16266         return (ill_flagaddr_cnt(ill, IPIF_DHCPRUNNING | IPIF_ADDRCONF,
16267             IPIF_NOFAILOVER));
16268 }

16270 /*
16271  * Return the number of point-to-point addresses on 'ill'.
16272  */
16273 uint_t
16274 ill_ptpaddr_cnt(const ill_t *ill)
16275 {
16276         return (ill_flagaddr_cnt(ill, IPIF_POINTOPOINT, 0));
16277 }

16279 /* ARGSUSED */
16280 int
16281 ip_sioctl_get_lifusesrc(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16282         ip_ioctl_cmd_t *ipip, void *ifreq)
16283 {
16284         struct lifreq   *lifr = ifreq;

16286         ASSERT(q->q_next == NULL);
16287         ASSERT(CONN_Q(q));

16289         ip1dbg(("ip_sioctl_get_lifusesrc(%s:%u %p)\n",
16290             ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
16291         lifr->lifr_index = ipif->ipif_ill->ill_usesrc_ifindex;
16292         ip1dbg(("ip_sioctl_get_lifusesrc:lifr_index = %d\n", lifr->lifr_index));

16294         return (0);
16295 }

16297 /* Find the previous ILL in this usesrc group */
```



```
16298 static ill_t *
16299 ill_prev_usesrc(ill_t *uill)
16300 {
16301         ill_t *ill;

16303         for (ill = uill->ill_usesrc_grp_next;
16304             ASSERT(ill), ill->ill_usesrc_grp_next != uill;
16305             ill = ill->ill_usesrc_grp_next)
16306                 /* do nothing */;
16307         return (ill);
16308 }

16310 /*
16311  * Release all members of the usesrc group. This routine is called
16312  * from ill_delete when the interface being unplumbed is the
16313  * group head.
16314  *
16315  * This silently clears the usesrc that ifconfig setup.
16316  * An alternative would be to keep that ifindex, and drop packets on the floor
16317  * since no source address can be selected.
16318  * Even if we keep the current semantics, don't need a lock and a linked list.
16319  * Can walk all the ills checking if they have a ill_usesrc_ifindex matching
16320  * the one that is being removed. Issue is how we return the usesrc users
16321  * (SIOCGLIFSRCOF). We want to be able to find the ills which have an
16322  * ill_usesrc_ifindex matching a target ill. We could also do that with an
16323  * ill walk, but the walker would need to insert in the ioctl response.
16324  */
16325 static void
16326 ill_disband_usesrc_group(ill_t *uill)
16327 {
16328         ill_t *next_ill, *tmp_ill;
16329         ip_stack_t      *ipst = uill->ill_ipst;

16331         ASSERT(RW_WRITE_HELD(&ipst->ips_ill_g_usesrc_lock));
16332         next_ill = uill->ill_usesrc_grp_next;

16334         do {
16335                 ASSERT(next_ill != NULL);
16336                 tmp_ill = next_ill->ill_usesrc_grp_next;
16337                 ASSERT(tmp_ill != NULL);
16338                 next_ill->ill_usesrc_grp_next = NULL;
16339                 next_ill->ill_usesrc_ifindex = 0;
16340                 next_ill = tmp_ill;
16341         } while (next_ill->ill_usesrc_ifindex != 0);
16342         uill->ill_usesrc_grp_next = NULL;
16343 }

16345 /*
16346  * Remove the client usesrc ILL from the list and relink to a new list
16347  */
16348 int
16349 ill_relink_usesrc_ills(ill_t *ucill, ill_t *uill, uint_t ifindex)
16350 {
16351         ill_t *ill, *tmp_ill;
16352         ip_stack_t      *ipst = ucill->ill_ipst;

16354         ASSERT((ucill != NULL) && (ucill->ill_usesrc_grp_next != NULL) &&
16355             (uill != NULL) && RW_WRITE_HELD(&ipst->ips_ill_g_usesrc_lock));

16357         /*
16358          * Check if the usesrc client ILL passed in is not already
16359          * in use as a usesrc ILL i.e one whose source address is
16360          * in use OR a usesrc ILL is not already in use as a usesrc
16361          * client ILL
16362          */
16363         if ((ucill->ill_usesrc_ifindex == 0) ||
```

```
16364             (uill->ill_usesrc_ifindex != 0)) {
16365                 return (-1);
16366         }

16368         ill = ill_prev_usesrc(ucill);
16369         ASSERT(ill->ill_usesrc_grp_next != NULL);

16371         /* Remove from the current list */
16372         if (ill->ill_usesrc_grp_next->ill_usesrc_grp_next == ill) {
16373                 /* Only two elements in the list */
16374                 ASSERT(ill->ill_usesrc_ifindex == 0);
16375                 ill->ill_usesrc_grp_next = NULL;
16376         } else {
16377                 ill->ill_usesrc_grp_next = ucill->ill_usesrc_grp_next;
16378         }

16380         if (ifindex == 0) {
16381                 ucill->ill_usesrc_ifindex = 0;
16382                 ucill->ill_usesrc_grp_next = NULL;
16383                 return (0);
16384         }

16386         ucill->ill_usesrc_ifindex = ifindex;
16387         tmp_ill = uill->ill_usesrc_grp_next;
16388         uill->ill_usesrc_grp_next = ucill;
16389         ucill->ill_usesrc_grp_next =
16390             (tmp_ill != NULL) ? tmp_ill : uill;
16391         return (0);
16392 }

16394 /*
16395  * Set the ill_usesrc and ill_usesrc_head fields. See synchronization notes in
16396  * ip.c for locking details.
16397  */
16398 /* ARGSUSED */
16399 int
16400 ip_sioctl_slifusesrc(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16401     ip_ioctl_cmd_t *ipip, void *ifreq)
16402 {
16403         struct lifreq *lifr = (struct lifreq *)ifreq;
16404         boolean_t isv6 = B_FALSE, reset_flg = B_FALSE;
16405         ill_t *usesrc_ill, *usesrc_cli_ill = ipif->ipif_ill;
16406         int err = 0, ret;
16407         uint_t ifindex;
16408         ipsq_t *ipsq = NULL;
16409         ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;

16411         ASSERT(IAM_WRITER_IPIF(ipif));
16412         ASSERT(q->q_next == NULL);
16413         ASSERT(CONN_Q(q));

16415         isv6 = (Q_TO_CONN(q))->conn_family == AF_INET6;

16417         ifindex = lifr->lifr_index;
16418         if (ifindex == 0) {
16419                 if (usesrc_cli_ill->ill_usesrc_grp_next == NULL) {
16420                         /* non usesrc group interface, nothing to reset */
16421                         return (0);
16422                 }
16423                 ifindex = usesrc_cli_ill->ill_usesrc_ifindex;
16424                 /* valid reset request */
16425                 reset_flg = B_TRUE;
16426         }

16428         usesrc_ill = ill_lookup_on_ifindex(ifindex, isv6, ipst);
16429         if (usesrc_ill == NULL)
```



```
16430                 return (ENXIO);
16431         if (usesrc_ill == ipif->ipif_ill) {
16432                 ill_refrele(usesrc_ill);
16433                 return (EINVAL);
16434         }

16436         ipsq = ipsq_try_enter(NULL, usesrc_ill, q, mp, ip_process_ioctl,
16437             NEW_OP, B_TRUE);
16438         if (ipsq == NULL) {
16439                 err = EINPROGRESS;
16440                 /* Operation enqueued on the ipsq of the usesrc ILL */
16441                 goto done;
16442         }

16444         /* USESRC isn't currently supported with IPMP */
16445         if (IS_IPMP(usesrc_ill) || IS_UNDER_IPMP(usesrc_ill)) {
16446                 err = ENOTSUP;
16447                 goto done;
16448         }

16450         /*
16451          * USESRC isn't compatible with the STANDBY flag.  (STANDBY is only
16452          * used by IPMP underlying interfaces, but someone might think it's
16453          * more general and try to use it independently with VNI.)
16454          */
16455         if (usesrc_ill->ill_phyint->phyint_flags & PHYI_STANDBY) {
16456                 err = ENOTSUP;
16457                 goto done;
16458         }

16460         /*
16461          * If the client is already in use as a usesrc_ill or a usesrc_ill is
16462          * already a client then return EINVAL
16463          */
16464         if (IS_USESRC_ILL(usesrc_cli_ill) || IS_USESRC_CLI_ILL(usesrc_ill)) {
16465                 err = EINVAL;
16466                 goto done;
16467         }

16469         /*
16470          * If the ill_usesrc_ifindex field is already set to what it needs to
16471          * be then this is a duplicate operation.
16472          */
16473         if (!reset_flg && usesrc_cli_ill->ill_usesrc_ifindex == ifindex) {
16474                 err = 0;
16475                 goto done;
16476         }

16478         ip1dbg(("ip_sioctl_slifusesrc: usesrc_cli_ill %s, usesrc_ill %s,"
16479             " v6 = %d", usesrc_cli_ill->ill_name, usesrc_ill->ill_name,
16480             usesrc_ill->ill_isv6));

16482         /*
16483          * ill_g_usesrc_lock global lock protects the ill_usesrc_grp_next
16484          * and the ill_usesrc_ifindex fields
16485          */
16486         rw_enter(&ipst->ips_ill_g_usesrc_lock, RW_WRITER);

16488         if (reset_flg) {
16489                 ret = ill_relink_usesrc_ills(usesrc_cli_ill, usesrc_ill, 0);
16490                 if (ret != 0) {
16491                         err = EINVAL;
16492                 }
16493                 rw_exit(&ipst->ips_ill_g_usesrc_lock);
16494                 goto done;
16495         }
```

```
16497         /*
16498          * Four possibilities to consider:
16499          * 1. Both usesrc_ill and usesrc_cli_ill are not part of any usesrc grp
16500          * 2. usesrc_ill is part of a group but usesrc_cli_ill isn't
16501          * 3. usesrc_cli_ill is part of a group but usesrc_ill isn't
16502          * 4. Both are part of their respective usesrc groups
16503          */
16504         if ((usesrc_ill->ill_usesrc_grp_next == NULL) &&
16505             (usesrc_cli_ill->ill_usesrc_grp_next == NULL)) {
16506                 ASSERT(usesrc_ill->ill_usesrc_ifindex == 0);
16507                 usesrc_cli_ill->ill_usesrc_ifindex = ifindex;
16508                 usesrc_ill->ill_usesrc_grp_next = usesrc_cli_ill;
16509                 usesrc_cli_ill->ill_usesrc_grp_next = usesrc_ill;
16510         } else if ((usesrc_ill->ill_usesrc_grp_next != NULL) &&
16511             (usesrc_cli_ill->ill_usesrc_grp_next == NULL)) {
16512                 usesrc_cli_ill->ill_usesrc_ifindex = ifindex;
16513                 /* Insert at head of list */
16514                 usesrc_cli_ill->ill_usesrc_grp_next =
16515                     usesrc_ill->ill_usesrc_grp_next;
16516                 usesrc_ill->ill_usesrc_grp_next = usesrc_cli_ill;
16517         } else {
16518                 ret = ill_relink_usesrc_ills(usesrc_cli_ill, usesrc_ill,
16519                     ifindex);
16520                 if (ret != 0)
16521                         err = EINVAL;
16522         }
16523         rw_exit(&ipst->ips_ill_g_usesrc_lock);

16525 done:
16526         if (ipsq != NULL)
16527                 ipsq_exit(ipsq);
16528         /* The refrele on the lifr_name ipif is done by ip_process_ioctl */
16529         ill_refrele(usesrc_ill);

16531         /* Let conn_ixa caching know that source address selection changed */
16532         ip_update_source_selection(ipst);

16534         return (err);
16535 }

16537 /* ARGSUSED */
16538 int
16539 ip_sioctl_get_dadstate(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16540     ip_ioctl_cmd_t *ipip, void *if_req)
16541 {
16542         struct lifreq   *lifr = (struct lifreq *)if_req;
16543         ill_t           *ill = ipif->ipif_ill;

16545         /*
16546          * Need a lock since IFF_UP can be set even when there are
16547          * references to the ipif.
16548          */
16549         mutex_enter(&ill->ill_lock);
16550         if ((ipif->ipif_flags & IPIF_UP) && ipif->ipif_addr_ready == 0)
16551                 lifr->lifr_dadstate = DAD_IN_PROGRESS;
16552         else
16553                 lifr->lifr_dadstate = DAD_DONE;
16554         mutex_exit(&ill->ill_lock);
16555         return (0);
16556 }

16558 /*
16559  * comparison function used by avl.
16560  */
16561 static int
```



```
16562 ill_phyint_compare_index(const void *index_ptr, const void *phyip)
16563 {

16565         uint_t index;

16567         ASSERT(phyip != NULL && index_ptr != NULL);

16569         index = *((uint_t *)index_ptr);
16570         /*
16571          * let the phyint with the lowest index be on top.
16572          */
16573         if (((phyint_t *)phyip)->phyint_ifindex < index)
16574                 return (1);
16575         if (((phyint_t *)phyip)->phyint_ifindex > index)
16576                 return (-1);
16577         return (0);
16578 }

16580 /*
16581  * comparison function used by avl.
16582  */
16583 static int
16584 ill_phyint_compare_name(const void *name_ptr, const void *phyip)
16585 {
16586         ill_t *ill;
16587         int res = 0;

16589         ASSERT(phyip != NULL && name_ptr != NULL);

16591         if (((phyint_t *)phyip)->phyint_illv4)
16592                 ill = ((phyint_t *)phyip)->phyint_illv4;
16593         else
16594                 ill = ((phyint_t *)phyip)->phyint_illv6;
16595         ASSERT(ill != NULL);

16597         res = strcmp(ill->ill_name, (char *)name_ptr);
16598         if (res > 0)
16599                 return (1);
16600         else if (res < 0)
16601                 return (-1);
16602         return (0);
16603 }

16605 /*
16606  * This function is called on the unplumb path via ill_glist_delete() when
16607  * there are no ills left on the phyint and thus the phyint can be freed.
16608  */
16609 static void
16610 phyint_free(phyint_t *phyi)
16611 {
16612         ip_stack_t *ipst = PHYINT_TO_IPST(phyi);

16614         ASSERT(phyi->phyint_illv4 == NULL && phyi->phyint_illv6 == NULL);

16616         /*
16617          * If this phyint was an IPMP meta-interface, blow away the group.
16618          * This is safe to do because all of the illgrps have already been
16619          * removed by I_PUNLINK, and thus SIOCSLIFGROUPNAME cannot find us.
16620          * If we're cleaning up as a result of failed initialization,
16621          * phyint_grp may be NULL.
16622          */
16623         if ((phyi->phyint_flags & PHYI_IPMP) && (phyi->phyint_grp != NULL)) {
16624                 rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);
16625                 ipmp_grp_destroy(phyi->phyint_grp);
16626                 phyi->phyint_grp = NULL;
16627                 rw_exit(&ipst->ips_ipmp_lock);
```

```
16628         }

16630         /*
16631          * If this interface was under IPMP, take it out of the group.
16632          */
16633         if (phyi->phyint_grp != NULL)
16634                 ipmp_phyint_leave_grp(phyi);

16636         /*
16637          * Delete the phyint and disassociate its ipsq.  The ipsq itself
16638          * will be freed in ipsq_exit().
16639          */
16640         phyi->phyint_ipsq->ipsq_phyint = NULL;
16641         phyi->phyint_name[0] = '\0';

16643         mi_free(phyi);
16644 }

16646 /*
16647  * Attach the ill to the phyint structure which can be shared by both
16648  * IPv4 and IPv6 ill. ill_init allocates a phyint to just hold flags. This
16649  * function is called from ipif_set_values and ill_lookup_on_name (for
16650  * loopback) where we know the name of the ill. We lookup the ill and if
16651  * there is one present already with the name use that phyint. Otherwise
16652  * reuse the one allocated by ill_init.
16653  */
16654 static void
16655 ill_phyint_reinit(ill_t *ill)
16656 {
16657         boolean_t isv6 = ill->ill_isv6;
16658         phyint_t *phyi_old;
16659         phyint_t *phyi;
16660         avl_index_t where = 0;
16661         ill_t   *ill_other = NULL;
16662         ip_stack_t      *ipst = ill->ill_ipst;

16664         ASSERT(RW_WRITE_HELD(&ipst->ips_ill_g_lock));

16666         phyi_old = ill->ill_phyint;
16667         ASSERT(isv6 || (phyi_old->phyint_illv4 == ill &&
16668             phyi_old->phyint_illv6 == NULL));
16669         ASSERT(!isv6 || (phyi_old->phyint_illv6 == ill &&
16670             phyi_old->phyint_illv4 == NULL));
16671         ASSERT(phyi_old->phyint_ifindex == 0);

16673         /*
16674          * Now that our ill has a name, set it in the phyint.
16675          */
16676         (void) strlcpy(ill->ill_phyint->phyint_name, ill->ill_name, LIFNAMSIZ);

16678         phyi = avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
16679             ill->ill_name, &where);

16681         /*
16682          * 1. We grabbed the ill_g_lock before inserting this ill into
16683          *    the global list of ills. So no other thread could have located
16684          *    this ill and hence the ipsq of this ill is guaranteed to be empty.
16685          * 2. Now locate the other protocol instance of this ill.
16686          * 3. Now grab both ill locks in the right order, and the phyint lock of
16687          *    the new ipsq. Holding ill locks + ill_g_lock ensures that the ipsq
16688          *    of neither ill can change.
16689          * 4. Merge the phyint and thus the ipsq as well of this ill onto the
16690          *    other ill.
16691          * 5. Release all locks.
16692          */
```



```
16694         /*
16695          * Look for IPv4 if we are initializing IPv6 or look for IPv6 if
16696          * we are initializing IPv4.
16697          */
16698         if (phyi != NULL) {
16699                 ill_other = (isv6) ? phyi->phyint_illv4 : phyi->phyint_illv6;
16700                 ASSERT(ill_other->ill_phyint != NULL);
16701                 ASSERT((isv6 && !ill_other->ill_isv6) ||
16702                     (!isv6 && ill_other->ill_isv6));
16703                 GRAB_ILL_LOCKS(ill, ill_other);
16704                 /*
16705                  * We are potentially throwing away phyint_flags which
16706                  * could be different from the one that we obtain from
16707                  * ill_other->ill_phyint. But it is okay as we are assuming
16708                  * that the state maintained within IP is correct.
16709                  */
16710                 mutex_enter(&phyi->phyint_lock);
16711                 if (isv6) {
16712                         ASSERT(phyi->phyint_illv6 == NULL);
16713                         phyi->phyint_illv6 = ill;
16714                 } else {
16715                         ASSERT(phyi->phyint_illv4 == NULL);
16716                         phyi->phyint_illv4 = ill;
16717                 }

16719                 /*
16720                  * Delete the old phyint and make its ipsq eligible
16721                  * to be freed in ipsq_exit().
16722                  */
16723                 phyi_old->phyint_illv4 = NULL;
16724                 phyi_old->phyint_illv6 = NULL;
16725                 phyi_old->phyint_ipsq->ipsq_phyint = NULL;
16726                 phyi_old->phyint_name[0] = '\0';
16727                 mi_free(phyi_old);
16728         } else {
16729                 mutex_enter(&ill->ill_lock);
16730                 /*
16731                  * We don't need to acquire any lock, since
16732                  * the ill is not yet visible globally  and we
16733                  * have not yet released the ill_g_lock.
16734                  */
16735                 phyi = phyi_old;
16736                 mutex_enter(&phyi->phyint_lock);
16737                 /* XXX We need a recovery strategy here. */
16738                 if (!phyint_assign_ifindex(phyi, ipst))
16739                         cmn_err(CE_PANIC, "phyint_assign_ifindex() failed");

16741                 avl_insert(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
16742                     (void *)phyi, where);

16744                 (void) avl_find(&ipst->ips_phyint_g_list->
16745                     phyint_list_avl_by_index,
16746                     &phyi->phyint_ifindex, &where);
16747                 avl_insert(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
16748                     (void *)phyi, where);
16749         }

16751         /*
16752          * Reassigning ill_phyint automatically reassigns the ipsq also.
16753          * pending mp is not affected because that is per ill basis.
16754          */
16755         ill->ill_phyint = phyi;

16757         /*
16758          * Now that the phyint's ifindex has been assigned, complete the
16759          * remaining
```

```
16760          */
16761         ill->ill_ip_mib->ipIfStatsIfIndex = ill->ill_phyint->phyint_ifindex;
16762         if (ill->ill_isv6) {
16763                 ill->ill_icmp6_mib->ipv6IfIcmpIfIndex =
16764                     ill->ill_phyint->phyint_ifindex;
16765                 ill->ill_mcast_type = ipst->ips_mld_max_version;
16766         } else {
16767                 ill->ill_mcast_type = ipst->ips_igmp_max_version;
16768         }

16770         /*
16771          * Generate an event within the hooks framework to indicate that
16772          * a new interface has just been added to IP.  For this event to
16773          * be generated, the network interface must, at least, have an
16774          * ifindex assigned to it.  (We don't generate the event for
16775          * loopback since ill_lookup_on_name() has its own NE_PLUMB event.)
16776          *
16777          * This needs to be run inside the ill_g_lock perimeter to ensure
16778          * that the ordering of delivered events to listeners matches the
16779          * order of them in the kernel.
16780          */
16781         if (!IS_LOOPBACK(ill)) {
16782                 ill_nic_event_dispatch(ill, 0, NE_PLUMB, ill->ill_name,
16783                     ill->ill_name_length);
16784         }
16785         RELEASE_ILL_LOCKS(ill, ill_other);
16786         mutex_exit(&phyi->phyint_lock);
16787 }

16789 /*
16790  * Notify any downstream modules of the name of this interface.
16791  * An M_IOCTL is used even though we don't expect a successful reply.
16792  * Any reply message from the driver (presumably an M_IOCNAK) will
16793  * eventually get discarded somewhere upstream.  The message format is
16794  * simply an SIOCSLIFNAME ioctl just as might be sent from ifconfig
16795  * to IP.
16796  */
16797 static void
16798 ip_ifname_notify(ill_t *ill, queue_t *q)
16799 {
16800         mblk_t *mp1, *mp2;
16801         struct iocblk *iocp;
16802         struct lifreq *lifr;

16804         mp1 = mkiocb(SIOCSLIFNAME);
16805         if (mp1 == NULL)
16806                 return;
16807         mp2 = allocb(sizeof (struct lifreq), BPRI_HI);
16808         if (mp2 == NULL) {
16809                 freeb(mp1);
16810                 return;
16811         }

16813         mp1->b_cont = mp2;
16814         iocp = (struct iocblk *)mp1->b_rptr;
16815         iocp->ioc_count = sizeof (struct lifreq);

16817         lifr = (struct lifreq *)mp2->b_rptr;
16818         mp2->b_wptr += sizeof (struct lifreq);
16819         bzero(lifr, sizeof (struct lifreq));

16821         (void) strncpy(lifr->lifr_name, ill->ill_name, LIFNAMSIZ);
16822         lifr->lifr_ppa = ill->ill_ppa;
16823         lifr->lifr_flags = (ill->ill_flags & (ILLF_IPV4|ILLF_IPV6));

16825         DTRACE_PROBE3(ill__dlpi, char *, "ip_ifname_notify",
```



```
16826             char *, "SIOCSLIFNAME", ill_t *, ill);
16827         putnext(q, mp1);
16828 }

16830 static int
16831 ipif_set_values_tail(ill_t *ill, ipif_t *ipif, mblk_t *mp, queue_t *q)
16832 {
16833         int             err;
16834         ip_stack_t      *ipst = ill->ill_ipst;
16835         phyint_t        *phyi = ill->ill_phyint;

16837         /*
16838          * Now that ill_name is set, the configuration for the IPMP
16839          * meta-interface can be performed.
16840          */
16841         if (IS_IPMP(ill)) {
16842                 rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);
16843                 /*
16844                  * If phyi->phyint_grp is NULL, then this is the first IPMP
16845                  * meta-interface and we need to create the IPMP group.
16846                  */
16847                 if (phyi->phyint_grp == NULL) {
16848                         /*
16849                          * If someone has renamed another IPMP group to have
16850                          * the same name as our interface, bail.
16851                          */
16852                         if (ipmp_grp_lookup(ill->ill_name, ipst) != NULL) {
16853                                 rw_exit(&ipst->ips_ipmp_lock);
16854                                 return (EEXIST);
16855                         }
16856                         phyi->phyint_grp = ipmp_grp_create(ill->ill_name, phyi);
16857                         if (phyi->phyint_grp == NULL) {
16858                                 rw_exit(&ipst->ips_ipmp_lock);
16859                                 return (ENOMEM);
16860                         }
16861                 }
16862                 rw_exit(&ipst->ips_ipmp_lock);
16863         }

16865         /* Tell downstream modules where they are. */
16866         ip_ifname_notify(ill, q);

16868         /*
16869          * ill_dl_phys returns EINPROGRESS in the usual case.
16870          * Error cases are ENOMEM ...
16871          */
16872         err = ill_dl_phys(ill, ipif, mp, q);

16874         if (ill->ill_isv6) {
16875                 mutex_enter(&ipst->ips_mld_slowtimeout_lock);
16876                 if (ipst->ips_mld_slowtimeout_id == 0) {
16877                         ipst->ips_mld_slowtimeout_id = timeout(mld_slowtimo,
16878                             (void *)ipst,
16879                             MSEC_TO_TICK(MCAST_SLOWTIMO_INTERVAL));
16880                 }
16881                 mutex_exit(&ipst->ips_mld_slowtimeout_lock);
16882         } else {
16883                 mutex_enter(&ipst->ips_igmp_slowtimeout_lock);
16884                 if (ipst->ips_igmp_slowtimeout_id == 0) {
16885                         ipst->ips_igmp_slowtimeout_id = timeout(igmp_slowtimo,
16886                             (void *)ipst,
16887                             MSEC_TO_TICK(MCAST_SLOWTIMO_INTERVAL));
16888                 }
16889                 mutex_exit(&ipst->ips_igmp_slowtimeout_lock);
16890         }
```

```
16892         return (err);
16893 }

16895 /*
16896  * Common routine for ppa and ifname setting. Should be called exclusive.
16897  *
16898  * Returns EINPROGRESS when mp has been consumed by queueing it on
16899  * ipx_pending_mp and the ioctl will complete in ip_rput.
16900  *
16901  * NOTE : If ppa is UNIT_MAX, we assign the next valid ppa and return
16902  * the new name and new ppa in lifr_name and lifr_ppa respectively.
16903  * For SLIFNAME, we pass these values back to the userland.
16904  */
16905 static int
16906 ipif_set_values(queue_t *q, mblk_t *mp, char *interf_name, uint_t *new_ppa_ptr)
16907 {
16908         ill_t   *ill;
16909         ipif_t  *ipif;
16910         ipsq_t  *ipsq;
16911         char    *ppa_ptr;
16912         char    *old_ptr;
16913         char    old_char;
16914         int     error;
16915         ip_stack_t      *ipst;

16917         ip1dbg(("ipif_set_values: interface %s\n", interf_name));
16918         ASSERT(q->q_next != NULL);
16919         ASSERT(interf_name != NULL);

16921         ill = (ill_t *)q->q_ptr;
16922         ipst = ill->ill_ipst;

16924         ASSERT(ill->ill_ipst != NULL);
16925         ASSERT(ill->ill_name[0] == '\0');
16926         ASSERT(IAM_WRITER_ILL(ill));
16927         ASSERT((mi_strlen(interf_name) + 1) <= LIFNAMSIZ);
16928         ASSERT(ill->ill_ppa == UINT_MAX);

16930         ill->ill_defend_start = ill->ill_defend_count = 0;
16931         /* The ppa is sent down by ifconfig or is chosen */
16932         if ((ppa_ptr = ill_get_ppa_ptr(interf_name)) == NULL) {
16933                 return (EINVAL);
16934         }

16936         /*
16937          * make sure ppa passed in is same as ppa in the name.
16938          * This check is not made when ppa == UINT_MAX in that case ppa
16939          * in the name could be anything. System will choose a ppa and
16940          * update new_ppa_ptr and inter_name to contain the choosen ppa.
16941          */
16942         if (*new_ppa_ptr != UINT_MAX) {
16943                 /* stoi changes the pointer */
16944                 old_ptr = ppa_ptr;
16945                 /*
16946                  * ifconfig passed in 0 for the ppa for DLPI 1 style devices
16947                  * (they don't have an externally visible ppa).  We assign one
16948                  * here so that we can manage the interface.  Note that in
16949                  * the past this value was always 0 for DLPI 1 drivers.
16950                  */
16951                 if (*new_ppa_ptr == 0)
16952                         *new_ppa_ptr = stoi(&old_ptr);
16953                 else if (*new_ppa_ptr != (uint_t)stoi(&old_ptr))
16954                         return (EINVAL);
16955         }
16956         /*
16957          * terminate string before ppa
```



```
16958          * save char at that location.
16959          */
16960         old_char = ppa_ptr[0];
16961         ppa_ptr[0] = '\0';

16963         ill->ill_ppa = *new_ppa_ptr;
16964         /*
16965          * Finish as much work now as possible before calling ill_glist_insert
16966          * which makes the ill globally visible and also merges it with the
16967          * other protocol instance of this phyint. The remaining work is
16968          * done after entering the ipsq which may happen sometime later.
16969          */
16970         ipif = ill->ill_ipif;

16972         /* We didn't do this when we allocated ipif in ip_ll_subnet_defaults */
16973         ipif_assign_seqid(ipif);

16975         if (!(ill->ill_flags & (ILLF_IPV4|ILLF_IPV6)))
16976                 ill->ill_flags |= ILLF_IPV4;

16978         ASSERT(ipif->ipif_next == NULL);        /* Only one ipif on ill */
16979         ASSERT((ipif->ipif_flags & IPIF_UP) == 0);

16981         if (ill->ill_flags & ILLF_IPV6) {

16983                 ill->ill_isv6 = B_TRUE;
16984                 ill_set_inputfn(ill);
16985                 if (ill->ill_rq != NULL) {
16986                         ill->ill_rq->q_qinfo = &iprinitv6;
16987                 }

16989                 /* Keep the !IN6_IS_ADDR_V4MAPPED assertions happy */
16990                 ipif->ipif_v6lcl_addr = ipv6_all_zeros;
16991                 ipif->ipif_v6subnet = ipv6_all_zeros;
16992                 ipif->ipif_v6net_mask = ipv6_all_zeros;
16993                 ipif->ipif_v6brd_addr = ipv6_all_zeros;
16994                 ipif->ipif_v6pp_dst_addr = ipv6_all_zeros;
16995                 ill->ill_reachable_retrans_time = ND_RETRANS_TIMER;
16996                 /*
16997                  * point-to-point or Non-mulicast capable
16998                  * interfaces won't do NUD unless explicitly
16999                  * configured to do so.
17000                  */
17001                 if (ipif->ipif_flags & IPIF_POINTOPOINT ||
17002                     !(ill->ill_flags & ILLF_MULTICAST)) {
17003                         ill->ill_flags |= ILLF_NONUD;
17004                 }
17005                 /* Make sure IPv4 specific flag is not set on IPv6 if */
17006                 if (ill->ill_flags & ILLF_NOARP) {
17007                         /*
17008                          * Note: xresolv interfaces will eventually need
17009                          * NOARP set here as well, but that will require
17010                          * those external resolvers to have some
17011                          * knowledge of that flag and act appropriately.
17012                          * Not to be changed at present.
17013                          */
17014                         ill->ill_flags &= ~ILLF_NOARP;
17015                 }
17016                 /*
17017                  * Set the ILLF_ROUTER flag according to the global
17018                  * IPv6 forwarding policy.
17019                  */
17020                 if (ipst->ips_ipv6_forwarding != 0)
17021                         ill->ill_flags |= ILLF_ROUTER;
17022         } else if (ill->ill_flags & ILLF_IPV4) {
17023                 ill->ill_isv6 = B_FALSE;
```

```
17024                   ill_set_inputfn(ill);
17025                   ill->ill_reachable_retrans_time = ARP_RETRANS_TIMER;
17026                   IN6_IPADDR_TO_V4MAPPED(INADDR_ANY, &ipif->ipif_v6lcl_addr);
17027                   IN6_IPADDR_TO_V4MAPPED(INADDR_ANY, &ipif->ipif_v6subnet);
17028                   IN6_IPADDR_TO_V4MAPPED(INADDR_ANY, &ipif->ipif_v6net_mask);
17029                   IN6_IPADDR_TO_V4MAPPED(INADDR_ANY, &ipif->ipif_v6brd_addr);
17030                   IN6_IPADDR_TO_V4MAPPED(INADDR_ANY, &ipif->ipif_v6pp_dst_addr);
17031                   /*
17032                    * Set the ILLF_ROUTER flag according to the global
17033                    * IPv4 forwarding policy.
17034                    */
17035                   if (ipst->ips_ip_forwarding != 0)
17036                           ill->ill_flags |= ILLF_ROUTER;
17037           }

17039           ASSERT(ill->ill_phyint != NULL);

17041           /*
17042            * The ipIfStatsIfindex and ipv6IfIcmpIfIndex assignments will
17043            * be completed in ill_glist_insert -> ill_phyint_reinit
17044            */
17045           if (!ill_allocate_mibs(ill))
17046                   return (ENOMEM);

17048           /*
17049            * Pick a default sap until we get the DL_INFO_ACK back from
17050            * the driver.
17051            */
17052           ill->ill_sap = (ill->ill_isv6) ? ill->ill_media->ip_m_ipv6sap :
17053               ill->ill_media->ip_m_ipv4sap;

17055           ill->ill_ifname_pending = 1;
17056           ill->ill_ifname_pending_err = 0;

17058           /*
17059            * When the first ipif comes up in ipif_up_done(), multicast groups
17060            * that were joined while this ill was not bound to the DLPI link need
17061            * to be recovered by ill_recover_multicast().
17062            */
17063           ill->ill_need_recover_multicast = 1;

17065           ill_refhold(ill);
17066           rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
17067           if ((error = ill_glist_insert(ill, interf_name,
17068               (ill->ill_flags & ILLF_IPV6) == ILLF_IPV6)) > 0) {
17069                   ill->ill_ppa = UINT_MAX;
17070                   ill->ill_name[0] = '\0';
17071                   /*
17072                    * undo null termination done above.
17073                    */
17074                   ppa_ptr[0] = old_char;
17075                   rw_exit(&ipst->ips_ill_g_lock);
17076                   ill_refrele(ill);
17077                   return (error);
17078           }

17080           ASSERT(ill->ill_name_length <= LIFNAMSIZ);

17082           /*
17083            * When we return the buffer pointed to by interf_name should contain
17084            * the same name as in ill_name.
17085            * If a ppa was choosen by the system (ppa passed in was UINT_MAX)
17086            * the buffer pointed to by new_ppa_ptr would not contain the right ppa
17087            * so copy full name and update the ppa ptr.
17088            * When ppa passed in != UINT_MAX all values are correct just undo
17089            * null termination, this saves a bcopy.
```



```
17090            */
17091           if (*new_ppa_ptr == UINT_MAX) {
17092                   bcopy(ill->ill_name, interf_name, ill->ill_name_length);
17093                   *new_ppa_ptr = ill->ill_ppa;
17094           } else {
17095                   /*
17096                    * undo null termination done above.
17097                    */
17098                   ppa_ptr[0] = old_char;
17099           }

17101           /* Let SCTP know about this ILL */
17102           sctp_update_ill(ill, SCTP_ILL_INSERT);

17104           /*
17105            * ill_glist_insert has made the ill visible globally, and
17106            * ill_phyint_reinit could have changed the ipsq. At this point,
17107            * we need to hold the ips_ill_g_lock across the call to enter the
17108            * ipsq to enforce atomicity and prevent reordering. In the event
17109            * the ipsq has changed, and if the new ipsq is currently busy,
17110            * we need to make sure that this half-completed ioctl is ahead of
17111            * any subsequent ioctl. We achieve this by not dropping the
17112            * ips_ill_g_lock which prevents any ill lookup itself thereby
17113            * ensuring that new ioctls can't start.
17114            */
17115           ipsq = ipsq_try_enter_internal(ill, q, mp, ip_reprocess_ioctl, NEW_OP,
17116               B_TRUE);

17118           rw_exit(&ipst->ips_ill_g_lock);
17119           ill_refrele(ill);
17120           if (ipsq == NULL)
17121                   return (EINPROGRESS);

17123           /*
17124            * If ill_phyint_reinit() changed our ipsq, then start on the new ipsq.
17125            */
17126           if (ipsq->ipsq_xop->ipx_current_ipif == NULL)
17127                   ipsq_current_start(ipsq, ipif, SIOCSLIFNAME);
17128           else
17129                   ASSERT(ipsq->ipsq_xop->ipx_current_ipif == ipif);

17131           error = ipif_set_values_tail(ill, ipif, mp, q);
17132           ipsq_exit(ipsq);
17133           if (error != 0 && error != EINPROGRESS) {
17134                   /*
17135                    * restore previous values
17136                    */
17137                   ill->ill_isv6 = B_FALSE;
17138                   ill_set_inputfn(ill);
17139           }
17140           return (error);
17141   }

17143   void
17144   ipif_init(ip_stack_t *ipst)
17145   {
17146           int i;

17148           for (i = 0; i < MAX_G_HEADS; i++) {
17149                   ipst->ips_ill_g_heads[i].ill_g_list_head =
17150                       (ill_if_t *)&ipst->ips_ill_g_heads[i];
17151                   ipst->ips_ill_g_heads[i].ill_g_list_tail =
17152                       (ill_if_t *)&ipst->ips_ill_g_heads[i];
17153           }

17155           avl_create(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
```

```
17156             ill_phyint_compare_index,
17157             sizeof (phyint_t),
17158             offsetof(struct phyint, phyint_avl_by_index));
17159         avl_create(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
17160             ill_phyint_compare_name,
17161             sizeof (phyint_t),
17162             offsetof(struct phyint, phyint_avl_by_name));
17163 }

17165 /*
17166  * Save enough information so that we can recreate the IRE if
17167  * the interface goes down and then up.
17168  */
17169 void
17170 ill_save_ire(ill_t *ill, ire_t *ire)
17171 {
17172         mblk_t  *save_mp;

17174         save_mp = allocb(sizeof (ifrt_t), BPRI_MED);
17175         if (save_mp != NULL) {
17176                 ifrt_t  *ifrt;

17178                 save_mp->b_wptr += sizeof (ifrt_t);
17179                 ifrt = (ifrt_t *)save_mp->b_rptr;
17180                 bzero(ifrt, sizeof (ifrt_t));
17181                 ifrt->ifrt_type = ire->ire_type;
17182                 if (ire->ire_ipversion == IPV4_VERSION) {
17183                         ASSERT(!ill->ill_isv6);
17184                         ifrt->ifrt_addr = ire->ire_addr;
17185                         ifrt->ifrt_gateway_addr = ire->ire_gateway_addr;
17186                         ifrt->ifrt_setsrc_addr = ire->ire_setsrc_addr;
17187                         ifrt->ifrt_mask = ire->ire_mask;
17188                 } else {
17189                         ASSERT(ill->ill_isv6);
17190                         ifrt->ifrt_v6addr = ire->ire_addr_v6;
17191                         /* ire_gateway_addr_v6 can change due to RTM_CHANGE */
17192                         mutex_enter(&ire->ire_lock);
17193                         ifrt->ifrt_v6gateway_addr = ire->ire_gateway_addr_v6;
17194                         mutex_exit(&ire->ire_lock);
17195                         ifrt->ifrt_v6setsrc_addr = ire->ire_setsrc_addr_v6;
17196                         ifrt->ifrt_v6mask = ire->ire_mask_v6;
17197                 }
17198                 ifrt->ifrt_flags = ire->ire_flags;
17199                 ifrt->ifrt_zoneid = ire->ire_zoneid;
17200                 mutex_enter(&ill->ill_saved_ire_lock);
17201                 save_mp->b_cont = ill->ill_saved_ire_mp;
17202                 ill->ill_saved_ire_mp = save_mp;
17203                 ill->ill_saved_ire_cnt++;
17204                 mutex_exit(&ill->ill_saved_ire_lock);
17205         }
17206 }

17208 /*
17209  * Remove one entry from ill_saved_ire_mp.
17210  */
17211 void
17212 ill_remove_saved_ire(ill_t *ill, ire_t *ire)
17213 {
17214         mblk_t  **mpp;
17215         mblk_t  *mp;
17216         ifrt_t  *ifrt;

17218         /* Remove from ill_saved_ire_mp list if it is there */
17219         mutex_enter(&ill->ill_saved_ire_lock);
17220         for (mpp = &ill->ill_saved_ire_mp; *mpp != NULL;
17221             mpp = &(*mpp)->b_cont) {
```



```
17222                 in6_addr_t      gw_addr_v6;

17224                 /*
17225                  * On a given ill, the tuple of address, gateway, mask,
17226                  * ire_type, and zoneid is unique for each saved IRE.
17227                  */
17228                 mp = *mpp;
17229                 ifrt = (ifrt_t *)mp->b_rptr;
17230                 /* ire_gateway_addr_v6 can change - need lock */
17231                 mutex_enter(&ire->ire_lock);
17232                 gw_addr_v6 = ire->ire_gateway_addr_v6;
17233                 mutex_exit(&ire->ire_lock);

17235                 if (ifrt->ifrt_zoneid != ire->ire_zoneid ||
17236                     ifrt->ifrt_type != ire->ire_type)
17237                         continue;

17239                 if (ill->ill_isv6 ?
17240                     (IN6_ARE_ADDR_EQUAL(&ifrt->ifrt_v6addr,
17241                     &ire->ire_addr_v6) &&
17242                     IN6_ARE_ADDR_EQUAL(&ifrt->ifrt_v6gateway_addr,
17243                     &gw_addr_v6) &&
17244                     IN6_ARE_ADDR_EQUAL(&ifrt->ifrt_v6mask,
17245                     &ire->ire_mask_v6)) :
17246                     (ifrt->ifrt_addr == ire->ire_addr &&
17247                     ifrt->ifrt_gateway_addr == ire->ire_gateway_addr &&
17248                     ifrt->ifrt_mask == ire->ire_mask)) {
17249                         *mpp = mp->b_cont;
17250                         ill->ill_saved_ire_cnt--;
17251                         freeb(mp);
17252                         break;
17253                 }
17254         }
17255         mutex_exit(&ill->ill_saved_ire_lock);
17256 }

17258 /*
17259  * IP multirouting broadcast routes handling
17260  * Append CGTP broadcast IREs to regular ones created
17261  * at ifconfig time.
17262  * The usage is a route add <cgtp_bc> <nic_bc> -multirt i.e., both
17263  * the destination and the gateway are broadcast addresses.
17264  * The caller has verified that the destination is an IRE_BROADCAST and that
17265  * RTF_MULTIRT was set. Here if the gateway is a broadcast address, then
17266  * we create a MULTIRT IRE_BROADCAST.
17267  * Note that the IRE_HOST created by ire_rt_add doesn't get found by anything
17268  * since the IRE_BROADCAST takes precedence; ire_add_v4 does head insertion.
17269  */
17270 static void
17271 ip_cgtp_bcast_add(ire_t *ire, ip_stack_t *ipst)
17272 {
17273         ire_t *ire_prim;

17275         ASSERT(ire != NULL);

17277         ire_prim = ire_ftable_lookup_v4(ire->ire_gateway_addr, 0, 0,
17278             IRE_BROADCAST, NULL, ALL_ZONES, NULL, MATCH_IRE_TYPE, 0, ipst,
17279             NULL);
17280         if (ire_prim != NULL) {
17281                 /*
17282                  * We are in the special case of broadcasts for
17283                  * CGTP. We add an IRE_BROADCAST that holds
17284                  * the RTF_MULTIRT flag, the destination
17285                  * address and the low level
17286                  * info of ire_prim. In other words, CGTP
17287                  * broadcast is added to the redundant ipif.
```

```
17288                 */
17289                ill_t *ill_prim;
17290                ire_t  *bcast_ire;

17292                ill_prim = ire_prim->ire_ill;

17294                ip2dbg(("ip_cgtp_filter_bcast_add: ire_prim %p, ill_prim %p\n",
17295                    (void *)ire_prim, (void *)ill_prim));

17297                bcast_ire = ire_create(
17298                    (uchar_t *)&ire->ire_addr,
17299                    (uchar_t *)&ip_g_all_ones,
17300                    (uchar_t *)&ire->ire_gateway_addr,
17301                    IRE_BROADCAST,
17302                    ill_prim,
17303                    GLOBAL_ZONEID,      /* CGTP is only for the global zone */
17304                    ire->ire_flags | RTF_KERNEL,
17305                    NULL,
17306                    ipst);

17308                /*
17309                 * Here we assume that ire_add does head insertion so that
17310                 * the added IRE_BROADCAST comes before the existing IRE_HOST.
17311                 */
17312                if (bcast_ire != NULL) {
17313                        if (ire->ire_flags & RTF_SETSRC) {
17314                                bcast_ire->ire_setsrc_addr =
17315                                    ire->ire_setsrc_addr;
17316                        }
17317                        bcast_ire = ire_add(bcast_ire);
17318                        if (bcast_ire != NULL) {
17319                                ip2dbg(("ip_cgtp_filter_bcast_add: "
17320                                    "added bcast_ire %p\n",
17321                                    (void *)bcast_ire));

17323                                ill_save_ire(ill_prim, bcast_ire);
17324                                ire_refrele(bcast_ire);
17325                        }
17326                }
17327                ire_refrele(ire_prim);
17328        }
17329 }

17331 /*
17332  * IP multirouting broadcast routes handling
17333  * Remove the broadcast ire.
17334  * The usage is a route delete <cgtp_bc> <nic_bc> -multirt i.e., both
17335  * the destination and the gateway are broadcast addresses.
17336  * The caller has only verified that RTF_MULTIRT was set. We check
17337  * that the destination is broadcast and that the gateway is a broadcast
17338  * address, and if so delete the IRE added by ip_cgtp_bcast_add().
17339  */
17340 static void
17341 ip_cgtp_bcast_delete(ire_t *ire, ip_stack_t *ipst)
17342 {
17343         ASSERT(ire != NULL);

17345         if (ip_type_v4(ire->ire_addr, ipst) == IRE_BROADCAST) {
17346                 ire_t *ire_prim;

17348                 ire_prim = ire_ftable_lookup_v4(ire->ire_gateway_addr, 0, 0,
17349                     IRE_BROADCAST, NULL, ALL_ZONES, NULL, MATCH_IRE_TYPE, 0,
17350                     ipst, NULL);
17351                 if (ire_prim != NULL) {
17352                         ill_t *ill_prim;
17353                         ire_t  *bcast_ire;
```



```
17355                         ill_prim = ire_prim->ire_ill;

17357                         ip2dbg(("ip_cgtp_filter_bcast_delete: "
17358                             "ire_prim %p, ill_prim %p\n",
17359                             (void *)ire_prim, (void *)ill_prim));

17361                         bcast_ire = ire_ftable_lookup_v4(ire->ire_addr, 0,
17362                             ire->ire_gateway_addr, IRE_BROADCAST,
17363                             ill_prim, ALL_ZONES, NULL,
17364                             MATCH_IRE_TYPE | MATCH_IRE_GW | MATCH_IRE_ILL |
17365                             MATCH_IRE_MASK, 0, ipst, NULL);

17367                         if (bcast_ire != NULL) {
17368                                 ip2dbg(("ip_cgtp_filter_bcast_delete: "
17369                                     "looked up bcast_ire %p\n",
17370                                     (void *)bcast_ire));
17371                                 ill_remove_saved_ire(bcast_ire->ire_ill,
17372                                     bcast_ire);
17373                                 ire_delete(bcast_ire);
17374                                 ire_refrele(bcast_ire);
17375                         }
17376                         ire_refrele(ire_prim);
17377                 }
17378         }
17379 }

17381 /*
17382  * Derive an interface id from the link layer address.
17383  * Knows about IEEE 802 and IEEE EUI-64 mappings.
17384  */
17385 static void
17386 ip_ether_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17387 {
17388         char            *addr;

17390         /*
17391          * Note that some IPv6 interfaces get plumbed over links that claim to
17392          * be DL_ETHER, but don't actually have Ethernet MAC addresses (e.g.
17393          * PPP links).  The ETHERADDRL check here ensures that we only set the
17394          * interface ID on IPv6 interfaces above links that actually have real
17395          * Ethernet addresses.
17396          */
17397         if (ill->ill_phys_addr_length == ETHERADDRL) {
17398                 /* Form EUI-64 like address */
17399                 addr = (char *)&v6addr->s6_addr32[2];
17400                 bcopy(ill->ill_phys_addr, addr, 3);
17401                 addr[0] ^= 0x2;         /* Toggle Universal/Local bit */
17402                 addr[3] = (char)0xff;
17403                 addr[4] = (char)0xfe;
17404                 bcopy(ill->ill_phys_addr + 3, addr + 5, 3);
17405         }
17406 }

17408 /* ARGSUSED */
17409 static void
17410 ip_nodef_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17411 {
17412 }

17414 typedef struct ipmp_ifcookie {
17415         uint32_t        ic_hostid;
17416         char            ic_ifname[LIFNAMSIZ];
17417         char            ic_zonename[ZONENAME_MAX];
17418 } ipmp_ifcookie_t;
```

```
17420 /*
17421  * Construct a pseudo-random interface ID for the IPMP interface that's both
17422  * predictable and (almost) guaranteed to be unique.
17423  */
17424 static void
17425 ip_ipmp_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17426 {
17427         zone_t          *zp;
17428         uint8_t         *addr;
17429         uchar_t         hash[16];
17430         ulong_t         hostid;
17431         MD5_CTX         ctx;
17432         ipmp_ifcookie_t ic = { 0 };

17434         ASSERT(IS_IPMP(ill));

17436         (void) ddi_strtoul(hw_serial, NULL, 10, &hostid);
17437         ic.ic_hostid = htonl((uint32_t)hostid);

17439         (void) strlcpy(ic.ic_ifname, ill->ill_name, LIFNAMSIZ);

17441         if ((zp = zone_find_by_id(ill->ill_zoneid)) != NULL) {
17442                 (void) strlcpy(ic.ic_zonename, zp->zone_name, ZONENAME_MAX);
17443                 zone_rele(zp);
17444         }

17446         MD5Init(&ctx);
17447         MD5Update(&ctx, &ic, sizeof (ic));
17448         MD5Final(hash, &ctx);

17450         /*
17451          * Map the hash to an interface ID per the basic approach in RFC3041.
17452          */
17453         addr = &v6addr->s6_addr8[8];
17454         bcopy(hash + 8, addr, sizeof (uint64_t));
17455         addr[0] &= ~0x2;                                /* set local bit */
17456 }

17458 /*
17459  * Map the multicast in6_addr_t in m_ip6addr to the physaddr for ethernet.
17460  */
17461 static void
17462 ip_ether_v6_mapping(ill_t *ill, uchar_t *m_ip6addr, uchar_t *m_physaddr)
17463 {
17464         phyint_t *phyi = ill->ill_phyint;

17466         /*
17467          * Check PHYI_MULTI_BCAST and length of physical
17468          * address to determine if we use the mapping or the
17469          * broadcast address.
17470          */
17471         if ((phyi->phyint_flags & PHYI_MULTI_BCAST) != 0 ||
17472             ill->ill_phys_addr_length != ETHERADDRL) {
17473                 ip_mbcast_mapping(ill, m_ip6addr, m_physaddr);
17474                 return;
17475         }
17476         m_physaddr[0] = 0x33;
17477         m_physaddr[1] = 0x33;
17478         m_physaddr[2] = m_ip6addr[12];
17479         m_physaddr[3] = m_ip6addr[13];
17480         m_physaddr[4] = m_ip6addr[14];
17481         m_physaddr[5] = m_ip6addr[15];
17482 }

17484 /*
17485  * Map the multicast ipaddr_t in m_ipaddr to the physaddr for ethernet.
```



```
17486  */
17487 static void
17488 ip_ether_v4_mapping(ill_t *ill, uchar_t *m_ipaddr, uchar_t *m_physaddr)
17489 {
17490         phyint_t *phyi = ill->ill_phyint;

17492         /*
17493          * Check PHYI_MULTI_BCAST and length of physical
17494          * address to determine if we use the mapping or the
17495          * broadcast address.
17496          */
17497         if ((phyi->phyint_flags & PHYI_MULTI_BCAST) != 0 ||
17498             ill->ill_phys_addr_length != ETHERADDRL) {
17499                 ip_mbcast_mapping(ill, m_ipaddr, m_physaddr);
17500                 return;
17501         }
17502         m_physaddr[0] = 0x01;
17503         m_physaddr[1] = 0x00;
17504         m_physaddr[2] = 0x5e;
17505         m_physaddr[3] = m_ipaddr[1] & 0x7f;
17506         m_physaddr[4] = m_ipaddr[2];
17507         m_physaddr[5] = m_ipaddr[3];
17508 }

17510 /* ARGSUSED */
17511 static void
17512 ip_mbcast_mapping(ill_t *ill, uchar_t *m_ipaddr, uchar_t *m_physaddr)
17513 {
17514         /*
17515          * for the MULTI_BCAST case and other cases when we want to
17516          * use the link-layer broadcast address for multicast.
17517          */
17518         uint8_t *bphys_addr;
17519         dl_unitdata_req_t *dlur;

17521         dlur = (dl_unitdata_req_t *)ill->ill_bcast_mp->b_rptr;
17522         if (ill->ill_sap_length < 0) {
17523                 bphys_addr = (uchar_t *)dlur +
17524                     dlur->dl_dest_addr_offset;
17525         } else  {
17526                 bphys_addr = (uchar_t *)dlur +
17527                     dlur->dl_dest_addr_offset + ill->ill_sap_length;
17528         }

17530         bcopy(bphys_addr, m_physaddr, ill->ill_phys_addr_length);
17531 }

17533 /*
17534  * Derive IPoIB interface id from the link layer address.
17535  */
17536 static void
17537 ip_ib_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17538 {
17539         char            *addr;

17541         ASSERT(ill->ill_phys_addr_length == 20);
17542         addr = (char *)&v6addr->s6_addr32[2];
17543         bcopy(ill->ill_phys_addr + 12, addr, 8);
17544         /*
17545          * In IBA 1.1 timeframe, some vendors erroneously set the u/l bit
17546          * in the globally assigned EUI-64 GUID to 1, in violation of IEEE
17547          * rules. In these cases, the IBA considers these GUIDs to be in
17548          * "Modified EUI-64" format, and thus toggling the u/l bit is not
17549          * required; vendors are required not to assign global EUI-64's
17550          * that differ only in u/l bit values, thus guaranteeing uniqueness
17551          * of the interface identifier. Whether the GUID is in modified
```

```
17552          * or proper EUI-64 format, the ipv6 identifier must have the u/l
17553          * bit set to 1.
17554          */
17555         addr[0] |= 2;                   /* Set Universal/Local bit to 1 */
17556 }

17558 /*
17559  * Map the multicast ipaddr_t in m_ipaddr to the physaddr for InfiniBand.
17560  * Note on mapping from multicast IP addresses to IPoIB multicast link
17561  * addresses. IPoIB multicast link addresses are based on IBA link addresses.
17562  * The format of an IPoIB multicast address is:
17563  *
17564  *  4 byte QPN      Scope Sign.  Pkey
17565  * +--------------------------------------------+
17566  * | 00FFFFFF | FF | 1X | X01B | Pkey | GroupID |
17567  * +--------------------------------------------+
17568  *
17569  * The Scope and Pkey components are properties of the IBA port and
17570  * network interface. They can be ascertained from the broadcast address.
17571  * The Sign. part is the signature, and is 401B for IPv4 and 601B for IPv6.
17572  */
17573 static void
17574 ip_ib_v4_mapping(ill_t *ill, uchar_t *m_ipaddr, uchar_t *m_physaddr)
17575 {
17576         static uint8_t ipv4_g_phys_ibmulti_addr[] = { 0x00, 0xff, 0xff, 0xff,
17577             0xff, 0x10, 0x40, 0x1b, 0x00, 0x00, 0x00, 0x00,
17578             0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
17579         uint8_t *bphys_addr;
17580         dl_unitdata_req_t *dlur;

17582         bcopy(ipv4_g_phys_ibmulti_addr, m_physaddr, ill->ill_phys_addr_length);

17584         /*
17585          * RFC 4391: IPv4 MGID is 28-bit long.
17586          */
17587         m_physaddr[16] = m_ipaddr[0] & 0x0f;
17588         m_physaddr[17] = m_ipaddr[1];
17589         m_physaddr[18] = m_ipaddr[2];
17590         m_physaddr[19] = m_ipaddr[3];


17593         dlur = (dl_unitdata_req_t *)ill->ill_bcast_mp->b_rptr;
17594         if (ill->ill_sap_length < 0) {
17595                 bphys_addr = (uchar_t *)dlur + dlur->dl_dest_addr_offset;
17596         } else  {
17597                 bphys_addr = (uchar_t *)dlur + dlur->dl_dest_addr_offset +
17598                     ill->ill_sap_length;
17599         }
17600         /*
17601          * Now fill in the IBA scope/Pkey values from the broadcast address.
17602          */
17603         m_physaddr[5] = bphys_addr[5];
17604         m_physaddr[8] = bphys_addr[8];
17605         m_physaddr[9] = bphys_addr[9];
17606 }

17608 static void
17609 ip_ib_v6_mapping(ill_t *ill, uchar_t *m_ipaddr, uchar_t *m_physaddr)
17610 {
17611         static uint8_t ipv4_g_phys_ibmulti_addr[] = { 0x00, 0xff, 0xff, 0xff,
17612             0xff, 0x10, 0x60, 0x1b, 0x00, 0x00, 0x00, 0x00,
17613             0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
17614         uint8_t *bphys_addr;
17615         dl_unitdata_req_t *dlur;

17617         bcopy(ipv4_g_phys_ibmulti_addr, m_physaddr, ill->ill_phys_addr_length);
```



```
17619         /*
17620          * RFC 4391: IPv4 MGID is 80-bit long.
17621          */
17622         bcopy(&m_ipaddr[6], &m_physaddr[10], 10);

17624         dlur = (dl_unitdata_req_t *)ill->ill_bcast_mp->b_rptr;
17625         if (ill->ill_sap_length < 0) {
17626                 bphys_addr = (uchar_t *)dlur + dlur->dl_dest_addr_offset;
17627         } else  {
17628                 bphys_addr = (uchar_t *)dlur + dlur->dl_dest_addr_offset +
17629                     ill->ill_sap_length;
17630         }
17631         /*
17632          * Now fill in the IBA scope/Pkey values from the broadcast address.
17633          */
17634         m_physaddr[5] = bphys_addr[5];
17635         m_physaddr[8] = bphys_addr[8];
17636         m_physaddr[9] = bphys_addr[9];
17637 }

17639 /*
17640  * Derive IPv6 interface id from an IPv4 link-layer address (e.g. from an IPv4
17641  * tunnel).  The IPv4 address simply get placed in the lower 4 bytes of the
17642  * IPv6 interface id.  This is a suggested mechanism described in section 3.7
17643  * of RFC4213.
17644  */
17645 static void
17646 ip_ipv4_genv6intfid(ill_t *ill, uint8_t *physaddr, in6_addr_t *v6addr)
17647 {
17648         ASSERT(ill->ill_phys_addr_length == sizeof (ipaddr_t));
17649         v6addr->s6_addr32[2] = 0;
17650         bcopy(physaddr, &v6addr->s6_addr32[3], sizeof (ipaddr_t));
17651 }

17653 /*
17654  * Derive IPv6 interface id from an IPv6 link-layer address (e.g. from an IPv6
17655  * tunnel).  The lower 8 bytes of the IPv6 address simply become the interface
17656  * id.
17657  */
17658 static void
17659 ip_ipv6_genv6intfid(ill_t *ill, uint8_t *physaddr, in6_addr_t *v6addr)
17660 {
17661         in6_addr_t *v6lladdr = (in6_addr_t *)physaddr;

17663         ASSERT(ill->ill_phys_addr_length == sizeof (in6_addr_t));
17664         bcopy(&v6lladdr->s6_addr32[2], &v6addr->s6_addr32[2], 8);
17665 }

17667 static void
17668 ip_ipv6_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17669 {
17670         ip_ipv6_genv6intfid(ill, ill->ill_phys_addr, v6addr);
17671 }

17673 static void
17674 ip_ipv6_v6destintfid(ill_t *ill, in6_addr_t *v6addr)
17675 {
17676         ip_ipv6_genv6intfid(ill, ill->ill_dest_addr, v6addr);
17677 }

17679 static void
17680 ip_ipv4_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17681 {
17682         ip_ipv4_genv6intfid(ill, ill->ill_phys_addr, v6addr);
17683 }
```

```
17685 static void
17686 ip_ipv4_v6destintfid(ill_t *ill, in6_addr_t *v6addr)
17687 {
17688         ip_ipv4_genv6intfid(ill, ill->ill_dest_addr, v6addr);
17689 }

17691 /*
17692  * Lookup an ill and verify that the zoneid has an ipif on that ill.
17693  * Returns an held ill, or NULL.
17694  */
17695 ill_t *
17696 ill_lookup_on_ifindex_zoneid(uint_t index, zoneid_t zoneid, boolean_t isv6,
17697     ip_stack_t *ipst)
17698 {
17699         ill_t   *ill;
17700         ipif_t  *ipif;

17702         ill = ill_lookup_on_ifindex(index, isv6, ipst);
17703         if (ill == NULL)
17704                 return (NULL);

17706         mutex_enter(&ill->ill_lock);
17707         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
17708                 if (IPIF_IS_CONDEMNED(ipif))
17709                         continue;
17710                 if (zoneid != ALL_ZONES && ipif->ipif_zoneid != zoneid &&
17711                     ipif->ipif_zoneid != ALL_ZONES)
17712                         continue;

17714                 mutex_exit(&ill->ill_lock);
17715                 return (ill);
17716         }
17717         mutex_exit(&ill->ill_lock);
17718         ill_refrele(ill);
17719         return (NULL);
17720 }

17722 /*
17723  * Return a pointer to an ipif_t given a combination of (ill_idx,ipif_id)
17724  * If a pointer to an ipif_t is returned then the caller will need to do
17725  * an ill_refrele().
17726  */
17727 ipif_t *
17728 ipif_getby_indexes(uint_t ifindex, uint_t lifidx, boolean_t isv6,
17729     ip_stack_t *ipst)
17730 {
17731         ipif_t *ipif;
17732         ill_t *ill;

17734         ill = ill_lookup_on_ifindex(ifindex, isv6, ipst);
17735         if (ill == NULL)
17736                 return (NULL);

17738         mutex_enter(&ill->ill_lock);
17739         if (ill->ill_state_flags & ILL_CONDEMNED) {
17740                 mutex_exit(&ill->ill_lock);
17741                 ill_refrele(ill);
17742                 return (NULL);
17743         }

17745         for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
17746                 if (!IPIF_CAN_LOOKUP(ipif))
17747                         continue;
17748                 if (lifidx == ipif->ipif_id) {
17749                         ipif_refhold_locked(ipif);
```



```
17750                         break;
17751                 }
17752         }

17754         mutex_exit(&ill->ill_lock);
17755         ill_refrele(ill);
17756         return (ipif);
17757 }

17759 /*
17760  * Set ill_inputfn based on the current know state.
17761  * This needs to be called when any of the factors taken into
17762  * account changes.
17763  */
17764 void
17765 ill_set_inputfn(ill_t *ill)
17766 {
17767         ip_stack_t      *ipst = ill->ill_ipst;

17769         if (ill->ill_isv6) {
17770                 if (is_system_labeled())
17771                         ill->ill_inputfn = ill_input_full_v6;
17772                 else
17773                         ill->ill_inputfn = ill_input_short_v6;
17774         } else {
17775                 if (is_system_labeled())
17776                         ill->ill_inputfn = ill_input_full_v4;
17777                 else if (ill->ill_dhcpinit != 0)
17778                         ill->ill_inputfn = ill_input_full_v4;
17779                 else if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_RSVP].connf_head
17780                     != NULL)
17781                         ill->ill_inputfn = ill_input_full_v4;
17782                 else if (ipst->ips_ip_cgtp_filter &&
17783                     ipst->ips_ip_cgtp_filter_ops != NULL)
17784                         ill->ill_inputfn = ill_input_full_v4;
17785                 else
17786                         ill->ill_inputfn = ill_input_short_v4;
17787         }
17788 }

17790 /*
17791  * Re-evaluate ill_inputfn for all the IPv4 ills.
17792  * Used when RSVP and CGTP comes and goes.
17793  */
17794 void
17795 ill_set_inputfn_all(ip_stack_t *ipst)
17796 {
17797         ill_walk_context_t      ctx;
17798         ill_t                   *ill;

17800         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
17801         ill = ILL_START_WALK_V4(&ctx, ipst);
17802         for (; ill != NULL; ill = ill_next(&ctx, ill))
17803                 ill_set_inputfn(ill);

17805         rw_exit(&ipst->ips_ill_g_lock);
17806 }

17808 /*
17809  * Set the physical address information for ‘ill’ to the contents of the
17810  * dl_notify_ind_t pointed to by ‘mp’.  Must be called as writer, and will be
17811  * asynchronous if ‘ill’ cannot immediately be quiesced -- in which case
17812  * EINPROGRESS will be returned.
17813  */
17814 int
17815 ill_set_phys_addr(ill_t *ill, mblk_t *mp)
```

```
17816 {
17817         ipsq_t *ipsq = ill->ill_phyint->phyint_ipsq;
17818         dl_notify_ind_t *dlindp = (dl_notify_ind_t *)mp->b_rptr;

17820         ASSERT(IAM_WRITER_IPSQ(ipsq));

17822         if (dlindp->dl_data != DL_IPV6_LINK_LAYER_ADDR &&
17823             dlindp->dl_data != DL_CURR_DEST_ADDR &&
17824             dlindp->dl_data != DL_CURR_PHYS_ADDR) {
17825                 /* Changing DL_IPV6_TOKEN is not yet supported */
17826                 return (0);
17827         }

17829         /*
17830          * We need to store up to two copies of 'mp' in 'ill'.  Due to the
17831          * design of ipsq_pending_mp_add(), we can't pass them as separate
17832          * arguments to ill_set_phys_addr_tail().  Instead, chain them
17833          * together here, then pull 'em apart in ill_set_phys_addr_tail().
17834          */
17835         if ((mp = copyb(mp)) == NULL || (mp->b_cont = copyb(mp)) == NULL) {
17836                 freemsg(mp);
17837                 return (ENOMEM);
17838         }

17840         ipsq_current_start(ipsq, ill->ill_ipif, 0);

17842         /*
17843          * Since we'll only do a logical down, we can't rely on ipif_down
17844          * to turn on ILL_DOWN_IN_PROGRESS, or for the DL_BIND_ACK to reset
17845          * ILL_DOWN_IN_PROGRESS. We instead manage this separately for this
17846          * case, to quiesce ire's and nce's for ill_is_quiescent.
17847          */
17848         mutex_enter(&ill->ill_lock);
17849         ill->ill_state_flags |= ILL_DOWN_IN_PROGRESS;
17850         /* no more ire/nce addition allowed */
17851         mutex_exit(&ill->ill_lock);

17853         /*
17854          * If we can quiesce the ill, then set the address.  If not, then
17855          * ill_set_phys_addr_tail() will be called from ipif_ill_refrele_tail().
17856          */
17857         ill_down_ipifs(ill, B_TRUE);
17858         mutex_enter(&ill->ill_lock);
17859         if (!ill_is_quiescent(ill)) {
17860                 /* call cannot fail since 'conn_t *' argument is NULL */
17861                 (void) ipsq_pending_mp_add(NULL, ill->ill_ipif, ill->ill_rq,
17862                     mp, ILL_DOWN);
17863                 mutex_exit(&ill->ill_lock);
17864                 return (EINPROGRESS);
17865         }
17866         mutex_exit(&ill->ill_lock);

17868         ill_set_phys_addr_tail(ipsq, ill->ill_rq, mp, NULL);
17869         return (0);
17870 }

17872 /*
17873  * When the allowed-ips link property is set on the datalink, IP receives a
17874  * DL_NOTE_ALLOWED_IPS notification that is processed in ill_set_allowed_ips()
17875  * to initialize the ill_allowed_ips[] array in the ill_t. This array is then
17876  * used to vet addresses passed to ip_sioctl_addr() and to ensure that the
17877  * only IP addresses configured on the ill_t are those in the ill_allowed_ips[]
17878  * array.
17879  */
17880 void
17881 ill_set_allowed_ips(ill_t *ill, mblk_t *mp)
```



```
17882 {
17883         ipsq_t *ipsq = ill->ill_phyint->phyint_ipsq;
17884         dl_notify_ind_t *dlip = (dl_notify_ind_t *)mp->b_rptr;
17885         mac_protect_t *mrp;
17886         int i;

17888         ASSERT(IAM_WRITER_IPSQ(ipsq));
17889         mrp = (mac_protect_t *)&dlip[1];

17891         if (mrp->mp_ipaddrcnt == 0) { /* reset allowed-ips */
17892                 kmem_free(ill->ill_allowed_ips,
17893                     ill->ill_allowed_ips_cnt * sizeof (in6_addr_t));
17894                 ill->ill_allowed_ips_cnt = 0;
17895                 ill->ill_allowed_ips = NULL;
17896                 mutex_enter(&ill->ill_phyint->phyint_lock);
17897                 ill->ill_phyint->phyint_flags &= ~PHYI_L3PROTECT;
17898                 mutex_exit(&ill->ill_phyint->phyint_lock);
17899                 return;
17900         }

17902         if (ill->ill_allowed_ips != NULL) {
17903                 kmem_free(ill->ill_allowed_ips,
17904                     ill->ill_allowed_ips_cnt * sizeof (in6_addr_t));
17905         }
17906         ill->ill_allowed_ips_cnt = mrp->mp_ipaddrcnt;
17907         ill->ill_allowed_ips = kmem_alloc(
17908             ill->ill_allowed_ips_cnt * sizeof (in6_addr_t), KM_SLEEP);
17909         for (i = 0; i < mrp->mp_ipaddrcnt;  i++)
17910                 ill->ill_allowed_ips[i] = mrp->mp_ipaddrs[i].ip_addr;

17912         mutex_enter(&ill->ill_phyint->phyint_lock);
17913         ill->ill_phyint->phyint_flags |= PHYI_L3PROTECT;
17914         mutex_exit(&ill->ill_phyint->phyint_lock);
17915 }

17917 /*
17918  * Once the ill associated with 'q' has quiesced, set its physical address
17919  * information to the values in 'addrmp'.  Note that two copies of 'addrmp'
17920  * are passed (linked by b_cont), since we sometimes need to save two distinct
17921  * copies in the ill_t, and our context doesn't permit sleeping or allocation
17922  * failure (we'll free the other copy if it's not needed).  Since the ill_t
17923  * is quiesced, we know any stale nce's with the old address information have
17924  * already been removed, so we don't need to call nce_flush().
17925  */
17926 /* ARGSUSED */
17927 static void
17928 ill_set_phys_addr_tail(ipsq_t *ipsq, queue_t *q, mblk_t *addrmp, void *dummy)
17929 {
17930         ill_t           *ill = q->q_ptr;
17931         mblk_t          *addrmp2 = unlinkb(addrmp);
17932         dl_notify_ind_t *dlindp = (dl_notify_ind_t *)addrmp->b_rptr;
17933         uint_t          addrlen, addroff;
17934         int             status;

17936         ASSERT(IAM_WRITER_IPSQ(ipsq));

17938         addroff = dlindp->dl_addr_offset;
17939         addrlen = dlindp->dl_addr_length - ABS(ill->ill_sap_length);

17941         switch (dlindp->dl_data) {
17942         case DL_IPV6_LINK_LAYER_ADDR:
17943                 ill_set_ndmp(ill, addrmp, addroff, addrlen);
17944                 freemsg(addrmp2);
17945                 break;

17947         case DL_CURR_DEST_ADDR:
```

```
17948                 freemsg(ill->ill_dest_addr_mp);
17949                 ill->ill_dest_addr = addrmp->b_rptr + addroff;
17950                 ill->ill_dest_addr_mp = addrmp;
17951                 if (ill->ill_isv6) {
17952                         ill_setdesttoken(ill);
17953                         ipif_setdestlinklocal(ill->ill_ipif);
17954                 }
17955                 freemsg(addrmp2);
17956                 break;

17958         case DL_CURR_PHYS_ADDR:
17959                 freemsg(ill->ill_phys_addr_mp);
17960                 ill->ill_phys_addr = addrmp->b_rptr + addroff;
17961                 ill->ill_phys_addr_mp = addrmp;
17962                 ill->ill_phys_addr_length = addrlen;
17963                 if (ill->ill_isv6)
17964                         ill_set_ndmp(ill, addrmp2, addroff, addrlen);
17965                 else
17966                         freemsg(addrmp2);
17967                 if (ill->ill_isv6) {
17968                         ill_setdefaulttoken(ill);
17969                         ipif_setlinklocal(ill->ill_ipif);
17970                 }
17971                 break;
17972         default:
17973                 ASSERT(0);
17974         }

17976         /*
17977          * reset ILL_DOWN_IN_PROGRESS so that we can successfully add ires
17978          * as we bring the ipifs up again.
17979          */
17980         mutex_enter(&ill->ill_lock);
17981         ill->ill_state_flags &= ~ILL_DOWN_IN_PROGRESS;
17982         mutex_exit(&ill->ill_lock);
17983         /*
17984          * If there are ipifs to bring up, ill_up_ipifs() will return
17985          * EINPROGRESS, and ipsq_current_finish() will be called by
17986          * ip_rput_dlpi_writer() or arp_bringup_done() when the last ipif is
17987          * brought up.
17988          */
17989         status = ill_up_ipifs(ill, q, addrmp);
17990         if (status != EINPROGRESS)
17991                 ipsq_current_finish(ipsq);
17992 }

17994 /*
17995  * Helper routine for setting the ill_nd_lla fields.
17996  */
17997 void
17998 ill_set_ndmp(ill_t *ill, mblk_t *ndmp, uint_t addroff, uint_t addrlen)
17999 {
18000         freemsg(ill->ill_nd_lla_mp);
18001         ill->ill_nd_lla = ndmp->b_rptr + addroff;
18002         ill->ill_nd_lla_mp = ndmp;
18003         ill->ill_nd_lla_len = addrlen;
18004 }

18006 /*
18007  * Replumb the ill.
18008  */
18009 int
18010 ill_replumb(ill_t *ill, mblk_t *mp)
18011 {
18012         ipsq_t *ipsq = ill->ill_phyint->phyint_ipsq;
```



```
18014         ASSERT(IAM_WRITER_IPSQ(ipsq));

18016         ipsq_current_start(ipsq, ill->ill_ipif, 0);

18018         /*
18019          * If we can quiesce the ill, then continue.  If not, then
18020          * ill_replumb_tail() will be called from ipif_ill_refrele_tail().
18021          */
18022         ill_down_ipifs(ill, B_FALSE);

18024         mutex_enter(&ill->ill_lock);
18025         if (!ill_is_quiescent(ill)) {
18026                 /* call cannot fail since `conn_t *' argument is NULL */
18027                 (void) ipsq_pending_mp_add(NULL, ill->ill_ipif, ill->ill_rq,
18028                     mp, ILL_DOWN);
18029                 mutex_exit(&ill->ill_lock);
18030                 return (EINPROGRESS);
18031         }
18032         mutex_exit(&ill->ill_lock);

18034         ill_replumb_tail(ipsq, ill->ill_rq, mp, NULL);
18035         return (0);
18036 }

18038 /* ARGSUSED */
18039 static void
18040 ill_replumb_tail(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy)
18041 {
18042         ill_t *ill = q->q_ptr;
18043         int err;
18044         conn_t *connp = NULL;

18046         ASSERT(IAM_WRITER_IPSQ(ipsq));
18047         freemsg(ill->ill_replumb_mp);
18048         ill->ill_replumb_mp = copyb(mp);

18050         if (ill->ill_replumb_mp == NULL) {
18051                 /* out of memory */
18052                 ipsq_current_finish(ipsq);
18053                 return;
18054         }

18056         mutex_enter(&ill->ill_lock);
18057         ill->ill_up_ipifs = ipsq_pending_mp_add(NULL, ill->ill_ipif,
18058             ill->ill_rq, ill->ill_replumb_mp, 0);
18059         mutex_exit(&ill->ill_lock);

18061         if (!ill->ill_up_ipifs) {
18062                 /* already closing */
18063                 ipsq_current_finish(ipsq);
18064                 return;
18065         }
18066         ill->ill_replumbing = 1;
18067         err = ill_down_ipifs_tail(ill);

18069         /*
18070          * Successfully quiesced and brought down the interface, now we send
18071          * the DL_NOTE_REPLUMB_DONE message down to the driver. Reuse the
18072          * DL_NOTE_REPLUMB message.
18073          */
18074         mp = mexchange(NULL, mp, sizeof (dl_notify_conf_t), M_PROTO,
18075             DL_NOTIFY_CONF);
18076         ASSERT(mp != NULL);
18077         ((dl_notify_conf_t *)mp->b_rptr)->dl_notification =
18078             DL_NOTE_REPLUMB_DONE;
18079         ill_dlpi_send(ill, mp);
```

```
18081         /*
18082          * For IPv4, we would usually get EINPROGRESS because the ETHERTYPE_ARP
18083          * streams have to be unbound. When all the DLPI exchanges are done,
18084          * ipsq_current_finish() will be called by arp_bringup_done(). The
18085          * remainder of ipif bringup via ill_up_ipifs() will also be done in
18086          * arp_bringup_done().
18087          */
18088         ASSERT(ill->ill_replumb_mp != NULL);
18089         if (err == EINPROGRESS)
18090                 return;
18091         else
18092                 ill->ill_replumb_mp = ipsq_pending_mp_get(ipsq, &connp);
18093         ASSERT(connp == NULL);
18094         if (err == 0 && ill->ill_replumb_mp != NULL &&
18095             ill_up_ipifs(ill, q, ill->ill_replumb_mp) == EINPROGRESS) {
18096                 return;
18097         }
18098         ipsq_current_finish(ipsq);
18099 }

18101 /*
18102  * Issue ioctl ‘cmd’ on ‘lh’; caller provides the initial payload in ‘buf’
18103  * which is ‘bufsize’ bytes.  On success, zero is returned and ‘buf’ updated
18104  * as per the ioctl.  On failure, an errno is returned.
18105  */
18106 static int
18107 ip_ioctl(ldi_handle_t lh, int cmd, void *buf, uint_t bufsize, cred_t *cr)
18108 {
18109         int rval;
18110         struct strioctl iocb;

18112         iocb.ic_cmd = cmd;
18113         iocb.ic_timout = 15;
18114         iocb.ic_len = bufsize;
18115         iocb.ic_dp = buf;

18117         return (ldi_ioctl(lh, I_STR, (intptr_t)&iocb, FKIOCTL, cr, &rval));
18118 }

18120 /*
18121  * Issue an SIOCGLIFCONF for address family ‘af’ and store the result into a
18122  * dynamically-allocated ‘lifcp’ that will be ‘bufsizep’ bytes on success.
18123  */
18124 static int
18125 ip_lifconf_ioctl(ldi_handle_t lh, int af, struct lifconf *lifcp,
18126     uint_t *bufsizep, cred_t *cr)
18127 {
18128         int err;
18129         struct lifnum lifn;

18131         bzero(&lifn, sizeof (lifn));
18132         lifn.lifn_family = af;
18133         lifn.lifn_flags = LIFC_UNDER_IPMP;

18135         if ((err = ip_ioctl(lh, SIOCGLIFNUM, &lifn, sizeof (lifn), cr)) != 0)
18136                 return (err);

18138         /*
18139          * Pad the interface count to account for additional interfaces that
18140          * may have been configured between the SIOCGLIFNUM and SIOCGLIFCONF.
18141          */
18142         lifn.lifn_count += 4;
18143         bzero(lifcp, sizeof (*lifcp));
18144         lifcp->lifc_flags = LIFC_UNDER_IPMP;
18145         lifcp->lifc_family = af;
```



```
18146         lifcp->lifc_len = *bufsizep = lifn.lifn_count * sizeof (struct lifreq);
18147         lifcp->lifc_buf = kmem_zalloc(*bufsizep, KM_SLEEP);

18149         err = ip_ioctl(lh, SIOCGLIFCONF, lifcp, sizeof (*lifcp), cr);
18150         if (err != 0) {
18151                 kmem_free(lifcp->lifc_buf, *bufsizep);
18152                 return (err);
18153         }

18155         return (0);
18156 }

18158 /*
18159  * Helper for ip_interface_cleanup() that removes the loopback interface.
18160  */
18161 static void
18162 ip_loopback_removeif(ldi_handle_t lh, boolean_t isv6, cred_t *cr)
18163 {
18164         int err;
18165         struct lifreq lifr;

18167         bzero(&lifr, sizeof (lifr));
18168         (void) strcpy(lifr.lifr_name, ipif_loopback_name);

18170         /*
18171          * Attempt to remove the interface.  It may legitimately not exist
18172          * (e.g. the zone administrator unplumbed it), so ignore ENXIO.
18173          */
18174         err = ip_ioctl(lh, SIOCLIFREMOVEIF, &lifr, sizeof (lifr), cr);
18175         if (err != 0 && err != ENXIO) {
18176                 ip0dbg(("ip_loopback_removeif: IP%s SIOCLIFREMOVEIF failed: "
18177                     "error %d\n", isv6 ? "v6" : "v4", err));
18178         }
18179 }

18181 /*
18182  * Helper for ip_interface_cleanup() that ensures no IP interfaces are in IPMP
18183  * groups and that IPMP data addresses are down.  These conditions must be met
18184  * so that IPMP interfaces can be I_PUNLINK’d, as per ip_sioctl_plink_ipmp().
18185  */
18186 static void
18187 ip_ipmp_cleanup(ldi_handle_t lh, boolean_t isv6, cred_t *cr)
18188 {
18189         int af = isv6 ? AF_INET6 : AF_INET;
18190         int i, nifs;
18191         int err;
18192         uint_t bufsize;
18193         uint_t lifrsize = sizeof (struct lifreq);
18194         struct lifconf lifc;
18195         struct lifreq *lifrp;

18197         if ((err = ip_lifconf_ioctl(lh, af, &lifc, &bufsize, cr)) != 0) {
18198                 cmn_err(CE_WARN, "ip_ipmp_cleanup: cannot get interface list "
18199                     "(error %d); any IPMP interfaces cannot be shutdown", err);
18200                 return;
18201         }

18203         nifs = lifc.lifc_len / lifrsize;
18204         for (lifrp = lifc.lifc_req, i = 0; i < nifs; i++, lifrp++) {
18205                 err = ip_ioctl(lh, SIOCGLIFFLAGS, lifrp, lifrsize, cr);
18206                 if (err != 0) {
18207                         cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot get "
18208                             "flags: error %d", lifrp->lifr_name, err);
18209                         continue;
18210                 }
```

```
18212                 if (lifrp->lifr_flags & IFF_IPMP) {
18213                         if ((lifrp->lifr_flags & (IFF_UP|IFF_DUPLICATE)) == 0)
18214                                 continue;

18216                         lifrp->lifr_flags &= ~IFF_UP;
18217                         err = ip_ioctl(lh, SIOCSLIFFLAGS, lifrp, lifrsize, cr);
18218                         if (err != 0) {
18219                                 cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot "
18220                                     "bring down (error %d); IPMP interface may "
18221                                     "not be shutdown", lifrp->lifr_name, err);
18222                         }

18224                         /*
18225                          * Check if IFF_DUPLICATE is still set -- and if so,
18226                          * reset the address to clear it.
18227                          */
18228                         err = ip_ioctl(lh, SIOCGLIFFLAGS, lifrp, lifrsize, cr);
18229                         if (err != 0 || !(lifrp->lifr_flags & IFF_DUPLICATE))
18230                                 continue;

18232                         err = ip_ioctl(lh, SIOCGLIFADDR, lifrp, lifrsize, cr);
18233                         if (err != 0 || (err = ip_ioctl(lh, SIOCGLIFADDR,
18234                             lifrp, lifrsize, cr)) != 0) {
18235                                 cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot "
18236                                     "reset DAD (error %d); IPMP interface may "
18237                                     "not be shutdown", lifrp->lifr_name, err);
18238                         }
18239                         continue;
18240                 }

18242                 if (strchr(lifrp->lifr_name, IPIF_SEPARATOR_CHAR) == 0) {
18243                         lifrp->lifr_groupname[0] = '\0';
18244                         if ((err = ip_ioctl(lh, SIOCSLIFGROUPNAME, lifrp,
18245                             lifrsize, cr)) != 0) {
18246                                 cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot "
18247                                     "leave IPMP group (error %d); associated "
18248                                     "IPMP interface may not be shutdown",
18249                                     lifrp->lifr_name, err);
18250                                 continue;
18251                         }
18252                 }
18253         }

18255         kmem_free(lifc.lifc_buf, bufsize);
18256 }

18258 #define UDPDEV          "/devices/pseudo/udp@0:udp"
18259 #define UDP6DEV         "/devices/pseudo/udp6@0:udp6"

18261 /*
18262  * Remove the loopback interfaces and prep the IPMP interfaces to be torn down.
18263  * Non-loopback interfaces are either I_LINK'd or I_PLINK'd; the former go away
18264  * when the user-level processes in the zone are killed and the latter are
18265  * cleaned up by str_stack_shutdown().
18266  */
18267 void
18268 ip_interface_cleanup(ip_stack_t *ipst)
18269 {
18270         ldi_handle_t    lh;
18271         ldi_ident_t     li;
18272         cred_t          *cr;
18273         int             err;
18274         int             i;
18275         char            *devs[] = { UDP6DEV, UDPDEV };
18276         netstackid_t    stackid = ipst->ips_netstack->netstack_stackid;
```



```
18278         if ((err = ldi_ident_from_major(ddi_name_to_major("ip"), &li)) != 0) {
18279                 cmn_err(CE_WARN, "ip_interface_cleanup: cannot get ldi ident:"
18280                     " error %d", err);
18281                 return;
18282         }

18284         cr = zone_get_kcred(netstackid_to_zoneid(stackid));
18285         ASSERT(cr != NULL);

18287         /*
18288          * NOTE: loop executes exactly twice and is hardcoded to know that the
18289          * first iteration is IPv6.  (Unrolling yields repetitious code, hence
18290          * the loop.)
18291          */
18292         for (i = 0; i < 2; i++) {
18293                 err = ldi_open_by_name(devs[i], FREAD|FWRITE, cr, &lh, li);
18294                 if (err != 0) {
18295                         cmn_err(CE_WARN, "ip_interface_cleanup: cannot open %s:"
18296                             " error %d", devs[i], err);
18297                         continue;
18298                 }

18300                 ip_loopback_removeif(lh, i == 0, cr);
18301                 ip_ipmp_cleanup(lh, i == 0, cr);

18303                 (void) ldi_close(lh, FREAD|FWRITE, cr);
18304         }

18306         ldi_ident_release(li);
18307         crfree(cr);
18308 }

18310 /*
18311  * This needs to be in-sync with nic_event_t definition
18312  */
18313 static const char *
18314 ill_hook_event2str(nic_event_t event)
18315 {
18316         switch (event) {
18317         case NE_PLUMB:
18318                 return ("PLUMB");
18319         case NE_UNPLUMB:
18320                 return ("UNPLUMB");
18321         case NE_UP:
18322                 return ("UP");
18323         case NE_DOWN:
18324                 return ("DOWN");
18325         case NE_ADDRESS_CHANGE:
18326                 return ("ADDRESS_CHANGE");
18327         case NE_LIF_UP:
18328                 return ("LIF_UP");
18329         case NE_LIF_DOWN:
18330                 return ("LIF_DOWN");
18331         case NE_IFINDEX_CHANGE:
18332                 return ("IFINDEX_CHANGE");
18333         default:
18334                 return ("UNKNOWN");
18335         }
18336 }

18338 void
18339 ill_nic_event_dispatch(ill_t *ill, lif_if_t lif, nic_event_t event,
18340     nic_event_data_t data, size_t datalen)
18341 {
18342         ip_stack_t              *ipst = ill->ill_ipst;
18343         hook_nic_event_int_t    *info;
```

```
18344         const char              *str = NULL;

18346         /* create a new nic event info */
18347         if ((info = kmem_alloc(sizeof (*info), KM_NOSLEEP)) == NULL)
18348                 goto fail;

18350         info->hnei_event.hne_nic = ill->ill_phyint->phyint_ifindex;
18351         info->hnei_event.hne_lif = lif;
18352         info->hnei_event.hne_event = event;
18353         info->hnei_event.hne_protocol = ill->ill_isv6 ?
18354             ipst->ips_ipv6_net_data : ipst->ips_ipv4_net_data;
18355         info->hnei_event.hne_data = NULL;
18356         info->hnei_event.hne_datalen = 0;
18357         info->hnei_stackid = ipst->ips_netstack->netstack_stackid;

18359         if (data != NULL && datalen != 0) {
18360                 info->hnei_event.hne_data = kmem_alloc(datalen, KM_NOSLEEP);
18361                 if (info->hnei_event.hne_data == NULL)
18362                         goto fail;
18363                 bcopy(data, info->hnei_event.hne_data, datalen);
18364                 info->hnei_event.hne_datalen = datalen;
18365         }

18367         if (ddi_taskq_dispatch(eventq_queue_nic, ip_ne_queue_func, info,
18368             DDI_NOSLEEP) == DDI_SUCCESS)
18369                 return;

18371 fail:
18372         if (info != NULL) {
18373                 if (info->hnei_event.hne_data != NULL) {
18374                         kmem_free(info->hnei_event.hne_data,
18375                             info->hnei_event.hne_datalen);
18376                 }
18377                 kmem_free(info, sizeof (hook_nic_event_t));
18378         }
18379         str = ill_hook_event2str(event);
18380         ip2dbg(("ill_nic_event_dispatch: could not dispatch %s nic event "
18381             "information for %s (ENOMEM)\n", str, ill->ill_name));
18382 }

18384 static int
18385 ipif_arp_up_done_tail(ipif_t *ipif, enum ip_resolver_action res_act)
18386 {
18387         int             err = 0;
18388         const in_addr_t *addr = NULL;
18389         nce_t           *nce = NULL;
18390         ill_t           *ill = ipif->ipif_ill;
18391         ill_t           *bound_ill;
18392         boolean_t       added_ipif = B_FALSE;
18393         uint16_t        state;
18394         uint16_t        flags;

18396         DTRACE_PROBE3(ipif__downup, char *, "ipif_arp_up_done_tail",
18397             ill_t *, ill, ipif_t *, ipif);
18398         if (ipif->ipif_lcl_addr != INADDR_ANY) {
18399                 addr = &ipif->ipif_lcl_addr;
18400         }

18402         if ((ipif->ipif_flags & IPIF_UNNUMBERED) || addr == NULL) {
18403                 if (res_act != Res_act_initial)
18404                         return (EINVAL);
18405         }

18407         if (addr != NULL) {
18408                 ipmp_illgrp_t   *illg = ill->ill_grp;
```



```
18410                 /* add unicast nce for the local addr */

18412                 if (IS_IPMP(ill)) {
18413                         /*
18414                          * If we're here via ipif_up(), then the ipif
18415                          * won't be bound yet -- add it to the group,
18416                          * which will bind it if possible. (We would
18417                          * add it in ipif_up(), but deleting on failure
18418                          * there is gruesome.)  If we're here via
18419                          * ipmp_ill_bind_ipif(), then the ipif has
18420                          * already been added to the group and we
18421                          * just need to use the binding.
18422                          */
18423                         if ((bound_ill = ipmp_ipif_bound_ill(ipif)) == NULL) {
18424                                 bound_ill  = ipmp_illgrp_add_ipif(illg, ipif);
18425                                 if (bound_ill == NULL) {
18426                                         /*
18427                                          * We couldn't bind the ipif to an ill
18428                                          * yet, so we have nothing to publish.
18429                                          * Mark the address as ready and return.
18430                                          */
18431                                         ipif->ipif_addr_ready = 1;
18432                                         return (0);
18433                                 }
18434                                 added_ipif = B_TRUE;
18435                         }
18436                 } else {
18437                         bound_ill = ill;
18438                 }

18440                 flags = (NCE_F_MYADDR | NCE_F_PUBLISH | NCE_F_AUTHORITY |
18441                     NCE_F_NONUD);
18442                 /*
18443                  * If this is an initial bring-up (or the ipif was never
18444                  * completely brought up), do DAD.  Otherwise, we're here
18445                  * because IPMP has rebound an address to this ill: send
18446                  * unsolicited advertisements (ARP announcements) to
18447                  * inform others.
18448                  */
18449                 if (res_act == Res_act_initial || !ipif->ipif_addr_ready) {
18450                         state = ND_UNCHANGED; /* compute in nce_add_common() */
18451                 } else {
18452                         state = ND_REACHABLE;
18453                         flags |= NCE_F_UNSOL_ADV;
18454                 }

18456 retry:
18457                 err = nce_lookup_then_add_v4(ill,
18458                     bound_ill->ill_phys_addr, bound_ill->ill_phys_addr_length,
18459                     addr, flags, state, &nce);

18461                 /*
18462                  * note that we may encounter EEXIST if we are moving
18463                  * the nce as a result of a rebind operation.
18464                  */
18465                 switch (err) {
18466                 case 0:
18467                         ipif->ipif_added_nce = 1;
18468                         nce->nce_ipif_cnt++;
18469                         break;
18470                 case EEXIST:
18471                         ip1dbg(("ipif_arp_up: NCE already exists for %s\n",
18472                             ill->ill_name));
18473                         if (!NCE_MYADDR(nce->nce_common)) {
18474                                 /*
18475                                  * A leftover nce from before this address
```

```
18476                                    * existed
18477                                    */
18478                                   ncec_delete(nce->nce_common);
18479                                   nce_refrele(nce);
18480                                   nce = NULL;
18481                                   goto retry;
18482                           }
18483                           if ((ipif->ipif_flags & IPIF_POINTOPOINT) == 0) {
18484                                   nce_refrele(nce);
18485                                   nce = NULL;
18486                                   ip1dbg(("ipif_arp_up: NCE already exists "
18487                                       "for %s:%u\n", ill->ill_name,
18488                                       ipif->ipif_id));
18489                                   goto arp_up_done;
18490                           }
18491                           /*
18492                            * Duplicate local addresses are permissible for
18493                            * IPIF_POINTOPOINT interfaces which will get marked
18494                            * IPIF_UNNUMBERED later in
18495                            * ip_addr_availability_check().
18496                            *
18497                            * The nce_ipif_cnt field tracks the number of
18498                            * ipifs that have nce_addr as their local address.
18499                            */
18500                           ipif->ipif_addr_ready = 1;
18501                           ipif->ipif_added_nce = 1;
18502                           nce->nce_ipif_cnt++;
18503                           err = 0;
18504                           break;
18505                   default:
18506                           ASSERT(nce == NULL);
18507                           goto arp_up_done;
18508                   }
18509                   if (arp_no_defense) {
18510                           if ((ipif->ipif_flags & IPIF_UP) &&
18511                               !ipif->ipif_addr_ready)
18512                                   ipif_up_notify(ipif);
18513                           ipif->ipif_addr_ready = 1;
18514                   }
18515           } else {
18516                   /* zero address. nothing to publish */
18517                   ipif->ipif_addr_ready = 1;
18518           }
18519           if (nce != NULL)
18520                   nce_refrele(nce);
18521   arp_up_done:
18522           if (added_ipif && err != 0)
18523                   ipmp_illgrp_del_ipif(ill->ill_grp, ipif);
18524           return (err);
18525   }

18527   int
18528   ipif_arp_up(ipif_t *ipif, enum ip_resolver_action res_act, boolean_t was_dup)
18529   {
18530           int             err = 0;
18531           ill_t           *ill = ipif->ipif_ill;
18532           boolean_t       first_interface, wait_for_dlpi = B_FALSE;

18534           DTRACE_PROBE3(ipif__downup, char *, "ipif_arp_up",
18535               ill_t *, ill, ipif_t *, ipif);

18537           /*
18538            * need to bring up ARP or setup mcast mapping only
18539            * when the first interface is coming UP.
18540            */
18541           first_interface = (ill->ill_ipif_up_count == 0 &&
```



```
18542               ill->ill_ipif_dup_count == 0 && !was_dup);

18544           if (res_act == Res_act_initial && first_interface) {
18545                   /*
18546                    * Send ATTACH + BIND
18547                    */
18548                   err = arp_ll_up(ill);
18549                   if (err != EINPROGRESS && err != 0)
18550                           return (err);

18552                   /*
18553                    * Add NCE for local address. Start DAD.
18554                    * we'll wait to hear that DAD has finished
18555                    * before using the interface.
18556                    */
18557                   if (err == EINPROGRESS)
18558                           wait_for_dlpi = B_TRUE;
18559           }

18561           if (!wait_for_dlpi)
18562                   (void) ipif_arp_up_done_tail(ipif, res_act);

18564           return (!wait_for_dlpi ? 0 : EINPROGRESS);
18565   }

18567   /*
18568    * Finish processing of "arp_up" after all the DLPI message
18569    * exchanges have completed between arp and the driver.
18570    */
18571   void
18572   arp_bringup_done(ill_t *ill, int err)
18573   {
18574           mblk_t  *mp1;
18575           ipif_t  *ipif;
18576           conn_t *connp = NULL;
18577           ipsq_t  *ipsq;
18578           queue_t *q;

18580           ip1dbg(("arp_bringup_done(%s)\n", ill->ill_name));

18582           ASSERT(IAM_WRITER_ILL(ill));

18584           ipsq = ill->ill_phyint->phyint_ipsq;
18585           ipif = ipsq->ipsq_xop->ipx_pending_ipif;
18586           mp1 = ipsq_pending_mp_get(ipsq, &connp);
18587           ASSERT(!((mp1 != NULL) ^ (ipif != NULL)));
18588           if (mp1 == NULL) /* bringup was aborted by the user */
18589                   return;

18591           /*
18592            * If an IOCTL is waiting on this (ipsq_current_ioctl != 0), then we
18593            * must have an associated conn_t.  Otherwise, we're bringing this
18594            * interface back up as part of handling an asynchronous event (e.g.,
18595            * physical address change).
18596            */
18597           if (ipsq->ipsq_xop->ipx_current_ioctl != 0) {
18598                   ASSERT(connp != NULL);
18599                   q = CONNP_TO_WQ(connp);
18600           } else {
18601                   ASSERT(connp == NULL);
18602                   q = ill->ill_rq;
18603           }
18604           if (err == 0) {
18605                   if (ipif->ipif_isv6) {
18606                           if ((err = ipif_up_done_v6(ipif)) != 0)
18607                                   ip0dbg(("arp_bringup_done: init failed\n"));
```

```
18608                 } else {
18609                         err = ipif_arp_up_done_tail(ipif, Res_act_initial);
18610                         if (err != 0 ||
18611                             (err = ipif_up_done(ipif)) != 0) {
18612                                 ip0dbg(("arp_bringup_done: "
18613                                     "init failed err %x\n", err));
18614                                 (void) ipif_arp_down(ipif);
18615                         }

18617                 }
18618         } else {
18619                 ip0dbg(("arp_bringup_done: DL_BIND_REQ failed\n"));
18620         }

18622         if ((err == 0) && (ill->ill_up_ipifs)) {
18623                 err = ill_up_ipifs(ill, q, mp1);
18624                 if (err == EINPROGRESS)
18625                         return;
18626         }

18628         /*
18629          * If we have a moved ipif to bring up, and everything has succeeded
18630          * to this point, bring it up on the IPMP ill.  Otherwise, leave it
18631          * down -- the admin can try to bring it up by hand if need be.
18632          */
18633         if (ill->ill_move_ipif != NULL) {
18634                 ipif = ill->ill_move_ipif;
18635                 ip1dbg(("bringing up ipif %p on ill %s\n", (void *)ipif,
18636                     ipif->ipif_ill->ill_name));
18637                 ill->ill_move_ipif = NULL;
18638                 if (err == 0) {
18639                         err = ipif_up(ipif, q, mp1);
18640                         if (err == EINPROGRESS)
18641                                 return;
18642                 }
18643         }

18645         /*
18646          * The operation must complete without EINPROGRESS since
18647          * ipsq_pending_mp_get() has removed the mblk from ipsq_pending_mp.
18648          * Otherwise, the operation will be stuck forever in the ipsq.
18649          */
18650         ASSERT(err != EINPROGRESS);
18651         if (ipsq->ipsq_xop->ipx_current_ioctl != 0) {
18652                 DTRACE_PROBE4(ipif__ioctl, char *, "arp_bringup_done finish",
18653                     int, ipsq->ipsq_xop->ipx_current_ioctl,
18654                     ill_t *, ill, ipif_t *, ipif);
18655                 ip_ioctl_finish(q, mp1, err, NO_COPYOUT, ipsq);
18656         } else {
18657                 ipsq_current_finish(ipsq);
18658         }
18659 }

18661 /*
18662  * Finish processing of arp replumb after all the DLPI message
18663  * exchanges have completed between arp and the driver.
18664  */
18665 void
18666 arp_replumb_done(ill_t *ill, int err)
18667 {
18668         mblk_t  *mp1;
18669         ipif_t  *ipif;
18670         conn_t *connp = NULL;
18671         ipsq_t  *ipsq;
18672         queue_t *q;
```



```
18674         ASSERT(IAM_WRITER_ILL(ill));

18676         ipsq = ill->ill_phyint->phyint_ipsq;
18677         ipif = ipsq->ipsq_xop->ipx_pending_ipif;
18678         mp1 = ipsq_pending_mp_get(ipsq, &connp);
18679         ASSERT(!((mp1 != NULL) ^ (ipif != NULL)));
18680         if (mp1 == NULL) {
18681                 ip0dbg(("arp_replumb_done: bringup aborted ioctl %x\n",
18682                     ipsq->ipsq_xop->ipx_current_ioctl));
18683                 /* bringup was aborted by the user */
18684                 return;
18685         }
18686         /*
18687          * If an IOCTL is waiting on this (ipsq_current_ioctl != 0), then we
18688          * must have an associated conn_t.  Otherwise, we're bringing this
18689          * interface back up as part of handling an asynchronous event (e.g.,
18690          * physical address change).
18691          */
18692         if (ipsq->ipsq_xop->ipx_current_ioctl != 0) {
18693                 ASSERT(connp != NULL);
18694                 q = CONNP_TO_WQ(connp);
18695         } else {
18696                 ASSERT(connp == NULL);
18697                 q = ill->ill_rq;
18698         }
18699         if ((err == 0) && (ill->ill_up_ipifs)) {
18700                 err = ill_up_ipifs(ill, q, mp1);
18701                 if (err == EINPROGRESS)
18702                         return;
18703         }
18704         /*
18705          * The operation must complete without EINPROGRESS since
18706          * ipsq_pending_mp_get() has removed the mblk from ipsq_pending_mp.
18707          * Otherwise, the operation will be stuck forever in the ipsq.
18708          */
18709         ASSERT(err != EINPROGRESS);
18710         if (ipsq->ipsq_xop->ipx_current_ioctl != 0) {
18711                 DTRACE_PROBE4(ipif__ioctl, char *,
18712                     "arp_replumb_done finish",
18713                     int, ipsq->ipsq_xop->ipx_current_ioctl,
18714                     ill_t *, ill, ipif_t *, ipif);
18715                 ip_ioctl_finish(q, mp1, err, NO_COPYOUT, ipsq);
18716         } else {
18717                 ipsq_current_finish(ipsq);
18718         }
18719 }

18721 void
18722 ipif_up_notify(ipif_t *ipif)
18723 {
18724         ip_rts_ifmsg(ipif, RTSQ_DEFAULT);
18725         ip_rts_newaddrmsg(RTM_ADD, 0, ipif, RTSQ_DEFAULT);
18726         sctp_update_ipif(ipif, SCTP_IPIF_UP);
18727         ill_nic_event_dispatch(ipif->ipif_ill, MAP_IPIF_ID(ipif->ipif_id),
18728             NE_LIF_UP, NULL, 0);
18729 }

18731 /*
18732  * ILB ioctl uses cv_wait (such as deleting a rule or adding a server) and
18733  * this assumes the context is cv_wait'able.  Hence it shouldnt' be used on
18734  * TPI end points with STREAMS modules pushed above.  This is assured by not
18735  * having the IPI_MODOK flag for the ioctl.  And IP ensures the ILB ioctl
18736  * never ends up on an ipsq, otherwise we may end up processing the ioctl
18737  * while unwinding from the ispq and that could be a thread from the bottom.
18738  */
18739 /* ARGSUSED */
```

```
18740 int
18741 ip_sioctl_ilb_cmd(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
18742     ip_ioctl_cmd_t *ipip, void *arg)
18743 {
18744         mblk_t *cmd_mp = mp->b_cont->b_cont;
18745         ilb_cmd_t command = *((ilb_cmd_t *)cmd_mp->b_rptr);
18746         int ret = 0;
18747         int i;
18748         size_t size;
18749         ip_stack_t *ipst;
18750         zoneid_t zoneid;
18751         ilb_stack_t *ilbs;

18753         ipst = CONNQ_TO_IPST(q);
18754         ilbs = ipst->ips_netstack->netstack_ilb;
18755         zoneid = Q_TO_CONN(q)->conn_zoneid;

18757         switch (command) {
18758         case ILB_CREATE_RULE: {
18759                 ilb_rule_cmd_t *cmd = (ilb_rule_cmd_t *)cmd_mp->b_rptr;

18761                 if (MBLKL(cmd_mp) != sizeof (ilb_rule_cmd_t)) {
18762                         ret = EINVAL;
18763                         break;
18764                 }

18766                 ret = ilb_rule_add(ilbs, zoneid, cmd);
18767                 break;
18768         }
18769         case ILB_DESTROY_RULE:
18770         case ILB_ENABLE_RULE:
18771         case ILB_DISABLE_RULE: {
18772                 ilb_name_cmd_t *cmd = (ilb_name_cmd_t *)cmd_mp->b_rptr;

18774                 if (MBLKL(cmd_mp) != sizeof (ilb_name_cmd_t)) {
18775                         ret = EINVAL;
18776                         break;
18777                 }

18779                 if (cmd->flags & ILB_RULE_ALLRULES) {
18780                         if (command == ILB_DESTROY_RULE) {
18781                                 ilb_rule_del_all(ilbs, zoneid);
18782                                 break;
18783                         } else if (command == ILB_ENABLE_RULE) {
18784                                 ilb_rule_enable_all(ilbs, zoneid);
18785                                 break;
18786                         } else if (command == ILB_DISABLE_RULE) {
18787                                 ilb_rule_disable_all(ilbs, zoneid);
18788                                 break;
18789                         }
18790                 } else {
18791                         if (command == ILB_DESTROY_RULE) {
18792                                 ret = ilb_rule_del(ilbs, zoneid, cmd->name);
18793                         } else if (command == ILB_ENABLE_RULE) {
18794                                 ret = ilb_rule_enable(ilbs, zoneid, cmd->name,
18795                                     NULL);
18796                         } else if (command == ILB_DISABLE_RULE) {
18797                                 ret = ilb_rule_disable(ilbs, zoneid, cmd->name,
18798                                     NULL);
18799                         }
18800                 }
18801                 break;
18802         }
18803         case ILB_NUM_RULES: {
18804                 ilb_num_rules_cmd_t *cmd;
```



```
18806                 if (MBLKL(cmd_mp) != sizeof (ilb_num_rules_cmd_t)) {
18807                         ret = EINVAL;
18808                         break;
18809                 }
18810                 cmd = (ilb_num_rules_cmd_t *)cmd_mp->b_rptr;
18811                 ilb_get_num_rules(ilbs, zoneid, &(cmd->num));
18812                 break;
18813         }
18814         case ILB_RULE_NAMES: {
18815                 ilb_rule_names_cmd_t *cmd;

18817                 cmd = (ilb_rule_names_cmd_t *)cmd_mp->b_rptr;
18818                 if (MBLKL(cmd_mp) < sizeof (ilb_rule_names_cmd_t) ||
18819                     cmd->num_names == 0) {
18820                         ret = EINVAL;
18821                         break;
18822                 }
18823                 size = cmd->num_names * ILB_RULE_NAMESZ;
18824                 if (cmd_mp->b_rptr + offsetof(ilb_rule_names_cmd_t, buf) +
18825                     size != cmd_mp->b_wptr) {
18826                         ret = EINVAL;
18827                         break;
18828                 }
18829                 ilb_get_rulenames(ilbs, zoneid, &cmd->num_names, cmd->buf);
18830                 break;
18831         }
18832         case ILB_NUM_SERVERS: {
18833                 ilb_num_servers_cmd_t *cmd;

18835                 if (MBLKL(cmd_mp) != sizeof (ilb_num_servers_cmd_t)) {
18836                         ret = EINVAL;
18837                         break;
18838                 }
18839                 cmd = (ilb_num_servers_cmd_t *)cmd_mp->b_rptr;
18840                 ret = ilb_get_num_servers(ilbs, zoneid, cmd->name,
18841                     &(cmd->num));
18842                 break;
18843         }
18844         case ILB_LIST_RULE: {
18845                 ilb_rule_cmd_t *cmd = (ilb_rule_cmd_t *)cmd_mp->b_rptr;

18847                 if (MBLKL(cmd_mp) != sizeof (ilb_rule_cmd_t)) {
18848                         ret = EINVAL;
18849                         break;
18850                 }
18851                 ret = ilb_rule_list(ilbs, zoneid, cmd);
18852                 break;
18853         }
18854         case ILB_LIST_SERVERS: {
18855                 ilb_servers_info_cmd_t *cmd;

18857                 cmd = (ilb_servers_info_cmd_t *)cmd_mp->b_rptr;
18858                 if (MBLKL(cmd_mp) < sizeof (ilb_servers_info_cmd_t) ||
18859                     cmd->num_servers == 0) {
18860                         ret = EINVAL;
18861                         break;
18862                 }
18863                 size = cmd->num_servers * sizeof (ilb_server_info_t);
18864                 if (cmd_mp->b_rptr + offsetof(ilb_servers_info_cmd_t, servers) +
18865                     size != cmd_mp->b_wptr) {
18866                         ret = EINVAL;
18867                         break;
18868                 }

18870                 ret = ilb_get_servers(ilbs, zoneid, cmd->name, cmd->servers,
18871                     &cmd->num_servers);
```

```
18872                   break;
18873           }
18874           case ILB_ADD_SERVERS: {
18875                   ilb_servers_info_cmd_t *cmd;
18876                   ilb_rule_t *rule;

18878                   cmd = (ilb_servers_info_cmd_t *)cmd_mp->b_rptr;
18879                   if (MBLKL(cmd_mp) < sizeof (ilb_servers_info_cmd_t)) {
18880                           ret = EINVAL;
18881                           break;
18882                   }
18883                   size = cmd->num_servers * sizeof (ilb_server_info_t);
18884                   if (cmd_mp->b_rptr + offsetof(ilb_servers_info_cmd_t, servers) +
18885                       size != cmd_mp->b_wptr) {
18886                           ret = EINVAL;
18887                           break;
18888                   }
18889                   rule = ilb_find_rule(ilbs, zoneid, cmd->name, &ret);
18890                   if (rule == NULL) {
18891                           ASSERT(ret != 0);
18892                           break;
18893                   }
18894                   for (i = 0; i < cmd->num_servers; i++) {
18895                           ilb_server_info_t *s;

18897                           s = &cmd->servers[i];
18898                           s->err = ilb_server_add(ilbs, rule, s);
18899                   }
18900                   ILB_RULE_REFRELE(rule);
18901                   break;
18902           }
18903           case ILB_DEL_SERVERS:
18904           case ILB_ENABLE_SERVERS:
18905           case ILB_DISABLE_SERVERS: {
18906                   ilb_servers_cmd_t *cmd;
18907                   ilb_rule_t *rule;
18908                   int (*f)();

18910                   cmd = (ilb_servers_cmd_t *)cmd_mp->b_rptr;
18911                   if (MBLKL(cmd_mp) < sizeof (ilb_servers_cmd_t)) {
18912                           ret = EINVAL;
18913                           break;
18914                   }
18915                   size = cmd->num_servers * sizeof (ilb_server_arg_t);
18916                   if (cmd_mp->b_rptr + offsetof(ilb_servers_cmd_t, servers) +
18917                       size != cmd_mp->b_wptr) {
18918                           ret = EINVAL;
18919                           break;
18920                   }

18922                   if (command == ILB_DEL_SERVERS)
18923                           f = ilb_server_del;
18924                   else if (command == ILB_ENABLE_SERVERS)
18925                           f = ilb_server_enable;
18926                   else if (command == ILB_DISABLE_SERVERS)
18927                           f = ilb_server_disable;

18929                   rule = ilb_find_rule(ilbs, zoneid, cmd->name, &ret);
18930                   if (rule == NULL) {
18931                           ASSERT(ret != 0);
18932                           break;
18933                   }

18935                   for (i = 0; i < cmd->num_servers; i++) {
18936                           ilb_server_arg_t *s;
```



```
18938                           s = &cmd->servers[i];
18939                           s->err = f(ilbs, zoneid, NULL, rule, &s->addr);
18940                   }
18941                   ILB_RULE_REFRELE(rule);
18942                   break;
18943           }
18944           case ILB_LIST_NAT_TABLE: {
18945                   ilb_list_nat_cmd_t *cmd;

18947                   cmd = (ilb_list_nat_cmd_t *)cmd_mp->b_rptr;
18948                   if (MBLKL(cmd_mp) < sizeof (ilb_list_nat_cmd_t)) {
18949                           ret = EINVAL;
18950                           break;
18951                   }
18952                   size = cmd->num_nat * sizeof (ilb_nat_entry_t);
18953                   if (cmd_mp->b_rptr + offsetof(ilb_list_nat_cmd_t, entries) +
18954                       size != cmd_mp->b_wptr) {
18955                           ret = EINVAL;
18956                           break;
18957                   }

18959                   ret = ilb_list_nat(ilbs, zoneid, cmd->entries, &cmd->num_nat,
18960                       &cmd->flags);
18961                   break;
18962           }
18963           case ILB_LIST_STICKY_TABLE: {
18964                   ilb_list_sticky_cmd_t *cmd;

18966                   cmd = (ilb_list_sticky_cmd_t *)cmd_mp->b_rptr;
18967                   if (MBLKL(cmd_mp) < sizeof (ilb_list_sticky_cmd_t)) {
18968                           ret = EINVAL;
18969                           break;
18970                   }
18971                   size = cmd->num_sticky * sizeof (ilb_sticky_entry_t);
18972                   if (cmd_mp->b_rptr + offsetof(ilb_list_sticky_cmd_t, entries) +
18973                       size != cmd_mp->b_wptr) {
18974                           ret = EINVAL;
18975                           break;
18976                   }

18978                   ret = ilb_list_sticky(ilbs, zoneid, cmd->entries,
18979                       &cmd->num_sticky, &cmd->flags);
18980                   break;
18981           }
18982           default:
18983                   ret = EINVAL;
18984                   break;
18985           }
18986   done:
18987           return (ret);
18988   }

18990   /* Remove all cache entries for this logical interface */
18991   void
18992   ipif_nce_down(ipif_t *ipif)
18993   {
18994           ill_t *ill = ipif->ipif_ill;
18995           nce_t *nce;

18997           DTRACE_PROBE3(ipif__downup, char *, "ipif_nce_down",
18998               ill_t *, ill, ipif_t *, ipif);
18999           if (ipif->ipif_added_nce) {
19000                   if (ipif->ipif_isv6)
19001                           nce = nce_lookup_v6(ill, &ipif->ipif_v6lcl_addr);
19002                   else
19003                           nce = nce_lookup_v4(ill, &ipif->ipif_lcl_addr);
```

```
19004                 if (nce != NULL) {
19005                         if (--nce->nce_ipif_cnt == 0)
19006                                 ncec_delete(nce->nce_common);
19007                         ipif->ipif_added_nce = 0;
19008                         nce_refrele(nce);
19009                 } else {
19010                         /*
19011                          * nce may already be NULL because it was already
19012                          * flushed, e.g., due to a call to nce_flush
19013                          */
19014                         ipif->ipif_added_nce = 0;
19015                 }
19016         }
19017         /*
19018          * Make IPMP aware of the deleted data address.
19019          */
19020         if (IS_IPMP(ill))
19021                 ipmp_illgrp_del_ipif(ill->ill_grp, ipif);

19023         /*
19024          * Remove all other nces dependent on this ill when the last ipif
19025          * is going away.
19026          */
19027         if (ill->ill_ipif_up_count == 0) {
19028                 ncec_walk(ill, (pfi_t)ncec_delete_per_ill,
19029                     (uchar_t *)ill, ill->ill_ipst);
19030                 if (IS_UNDER_IPMP(ill))
19031                         nce_flush(ill, B_TRUE);
19032         }
19033 }

19035 /*
19036  * find the first interface that uses usill for its source address.
19037  */
19038 ill_t *
19039 ill_lookup_usesrc(ill_t *usill)
19040 {
19041         ip_stack_t *ipst = usill->ill_ipst;
19042         ill_t *ill;

19044         ASSERT(usill != NULL);

19046         /* ill_g_usesrc_lock protects ill_usesrc_grp_next */
19047         rw_enter(&ipst->ips_ill_g_usesrc_lock, RW_WRITER);
19048         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
19049         for (ill = usill->ill_usesrc_grp_next; ill != NULL && ill != usill;
19050             ill = ill->ill_usesrc_grp_next) {
19051                 if (!IS_UNDER_IPMP(ill) && (ill->ill_flags & ILLF_MULTICAST) &&
19052                     !ILL_IS_CONDEMNED(ill)) {
19053                         ill_refhold(ill);
19054                         break;
19055                 }
19056         }
19057         rw_exit(&ipst->ips_ill_g_lock);
19058         rw_exit(&ipst->ips_ill_g_usesrc_lock);
19059         return (ill);
19060 }

19062 /*
19063  * This comment applies to both ip_sioctl_get_ifhwaddr and
19064  * ip_sioctl_get_lifhwaddr as the basic function of these two functions
19065  * is the same.
19066  *
19067  * The goal here is to find an IP interface that corresponds to the name
19068  * provided by the caller in the ifreq/lifreq structure held in the mblk_t
19069  * chain and to fill out a sockaddr/sockaddr_storage structure with the
```



```
19070  * mac address.
19071  *
19072  * The SIOCGIFHWADDR/SIOCGLIFHWADDR ioctl may return an error for a number
19073  * of different reasons:
19074  * ENXIO - the device name is not known to IP.
19075  * EADDRNOTAVAIL - the device has no hardware address. This is indicated
19076  * by ill_phys_addr not pointing to an actual address.
19077  * EPFNOSUPPORT - this will indicate that a request is being made for a
19078  * mac address that will not fit in the data structure supplier (struct
19079  * sockaddr).
19080  *
19081  */
19082 /* ARGSUSED */
19083 int
19084 ip_sioctl_get_ifhwaddr(ipif_t *ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
19085     ip_ioctl_cmd_t *ipip, void *if_req)
19086 {
19087         struct sockaddr *sock;
19088         struct ifreq *ifr;
19089         mblk_t *mp1;
19090         ill_t *ill;

19092         ASSERT(ipif != NULL);
19093         ill = ipif->ipif_ill;

19095         if (ill->ill_phys_addr == NULL) {
19096                 return (EADDRNOTAVAIL);
19097         }
19098         if (ill->ill_phys_addr_length > sizeof (sock->sa_data)) {
19099                 return (EPFNOSUPPORT);
19100         }

19102         ip1dbg(("ip_sioctl_get_hwaddr(%s)\n", ill->ill_name));

19104         /* Existence of mp1 has been checked in ip_wput_nondata */
19105         mp1 = mp->b_cont->b_cont;
19106         ifr = (struct ifreq *)mp1->b_rptr;

19108         sock = &ifr->ifr_addr;
19109         /*
19110          * The "family" field in the returned structure is set to a value
19111          * that represents the type of device to which the address belongs.
19112          * The value returned may differ to that on Linux but it will still
19113          * represent the correct symbol on Solaris.
19114          */
19115         sock->sa_family = arp_hw_type(ill->ill_mactype);
19116         bcopy(ill->ill_phys_addr, &sock->sa_data, ill->ill_phys_addr_length);

19118         return (0);
19119 }

19121 /*
19122  * The expection of applications using SIOCGIFHWADDR is that data will
19123  * be returned in the sa_data field of the sockaddr structure. With
19124  * SIOCGLIFHWADDR, we're breaking new ground as there is no Linux
19125  * equivalent. In light of this, struct sockaddr_dl is used as it
19126  * offers more space for address storage in sll_data.
19127  */
19128 /* ARGSUSED */
19129 int
19130 ip_sioctl_get_lifhwaddr(ipif_t *ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
19131     ip_ioctl_cmd_t *ipip, void *if_req)
19132 {
19133         struct sockaddr_dl *sock;
19134         struct lifreq *lifr;
19135         mblk_t *mp1;
```

```
19136         ill_t *ill;

19138         ASSERT(ipif != NULL);
19139         ill = ipif->ipif_ill;

19141         if (ill->ill_phys_addr == NULL) {
19142                 return (EADDRNOTAVAIL);
19143         }
19144         if (ill->ill_phys_addr_length > sizeof (sock->sdl_data)) {
19145                 return (EPFNOSUPPORT);
19146         }

19148         ip1dbg(("ip_sioctl_get_lifhwaddr(%s)\n", ill->ill_name));

19150         /* Existence of mp1 has been checked in ip_wput_nondata */
19151         mp1 = mp->b_cont->b_cont;
19152         lifr = (struct lifreq *)mp1->b_rptr;

19154         /*
19155          * sockaddr_ll is used here because it is also the structure used in
19156          * responding to the same ioctl in sockpfp. The only other choice is
19157          * sockaddr_dl which contains fields that are not required here
19158          * because its purpose is different.
19159          */
19160         lifr->lifr_type = ill->ill_type;
19161         sock = (struct sockaddr_dl *)&lifr->lifr_addr;
19162         sock->sdl_family = AF_LINK;
19163         sock->sdl_index = ill->ill_phyint->phyint_ifindex;
19164         sock->sdl_type = ill->ill_mactype;
19165         sock->sdl_nlen = 0;
19166         sock->sdl_slen = 0;
19167         sock->sdl_alen = ill->ill_phys_addr_length;
19168         bcopy(ill->ill_phys_addr, sock->sdl_data, ill->ill_phys_addr_length);

19170         return (0);
19171 }
```

```
**********************************************************
   89715 Mon Jul  9 14:38:17 2012
new/usr/src/uts/common/inet/ip/ip_input.c
dccp: ips_ipcl_dccp_fanout
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 *
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */
/* Copyright (c) 1990 Mentat Inc. */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/dlpi.h>
#include <sys/stropts.h>
#include <sys/sysmacros.h>
#include <sys/strsubr.h>
#include <sys/strlog.h>
#include <sys/strsun.h>
#include <sys/zone.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/xti_inet.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/kobj.h>
#include <sys/modctl.h>
#include <sys/atomic.h>
#include <sys/policy.h>
#include <sys/priv.h>

#include <sys/systm.h>
#include <sys/param.h>
#include <sys/kmem.h>
#include <sys/sdt.h>
#include <sys/socket.h>
#include <sys/vtrace.h>
#include <sys/isa_defs.h>
#include <sys/mac.h>
#include <net/if.h>
#include <net/if_arp.h>
#include <net/route.h>
```



```
#include <sys/sockio.h>
#include <netinet/in.h>
#include <net/if_dl.h>

#include <inet/common.h>
#include <inet/mi.h>
#include <inet/mib2.h>
#include <inet/nd.h>
#include <inet/arp.h>
#include <inet/snmpcom.h>
#include <inet/kstatcom.h>

#include <netinet/igmp_var.h>
#include <netinet/ip6.h>
#include <netinet/icmp6.h>
#include <netinet/sctp.h>

#include <inet/ip.h>
#include <inet/ip_impl.h>
#include <inet/ip6.h>
#include <inet/ip6_asp.h>
#include <inet/optcom.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>
#include <inet/ip_multi.h>
#include <inet/ip_if.h>
#include <inet/ip_ire.h>
#include <inet/ip_ftable.h>
#include <inet/ip_rts.h>
#include <inet/ip_ndp.h>
#include <inet/ip_listutils.h>
#include <netinet/igmp.h>
#include <netinet/ip_mroute.h>
#include <inet/ipp_common.h>

#include <net/pfkeyv2.h>
#include <inet/sadb.h>
#include <inet/ipsec_impl.h>
#include <inet/ipdrop.h>
#include <inet/ip_netinfo.h>
#include <inet/ilb_ip.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>

#include <sys/ethernet.h>
#include <net/if_types.h>
#include <sys/cpuvar.h>

#include <ipp/ipp.h>
#include <ipp/ipp_impl.h>
#include <ipp/ipgpc/ipgpc.h>

#include <sys/pattr.h>
#include <inet/ipclassifier.h>
#include <inet/sctp_ip.h>
#include <inet/sctp/sctp_impl.h>
#include <inet/udp_impl.h>
#include <inet/dccp/dccp_impl.h>
#endif /* ! codereview */
#include <sys/sunddi.h>

#include <sys/tsol/label.h>
#include <sys/tsol/tnet.h>

#include <sys/clock_impl.h>     /* For LBOLT_FASTPATH{,64} */
```

```
128 #ifdef  DEBUG
129 extern boolean_t skip_sctp_cksum;
130 #endif

132 static void     ip_input_local_v4(ire_t *, mblk_t *, ipha_t *,
133     ip_recv_attr_t *);

135 static void     ip_input_broadcast_v4(ire_t *, mblk_t *, ipha_t *,
136     ip_recv_attr_t *);
137 static void     ip_input_multicast_v4(ire_t *, mblk_t *, ipha_t *,
138     ip_recv_attr_t *);

140 #pragma inline(ip_input_common_v4, ip_input_local_v4, ip_forward_xmit_v4)

142 /*
143  * Direct read side procedure capable of dealing with chains. GLDv3 based
144  * drivers call this function directly with mblk chains while STREAMS
145  * read side procedure ip_rput() calls this for single packet with ip_ring
146  * set to NULL to process one packet at a time.
147  *
148  * The ill will always be valid if this function is called directly from
149  * the driver.
150  *
151  * If ip_input() is called from GLDv3:
152  *
153  *   - This must be a non-VLAN IP stream.
154  *   - 'mp' is either an untagged or a special priority-tagged packet.
155  *   - Any VLAN tag that was in the MAC header has been stripped.
156  *
157  * If the IP header in packet is not 32-bit aligned, every message in the
158  * chain will be aligned before further operations. This is required on SPARC
159  * platform.
160  */
161 void
162 ip_input(ill_t *ill, ill_rx_ring_t *ip_ring, mblk_t *mp_chain,
163     struct mac_header_info_s *mhip)
164 {
165         (void) ip_input_common_v4(ill, ip_ring, mp_chain, mhip, NULL, NULL,
166             NULL);
167 }

169 /*
170  * ip_accept_tcp() - This function is called by the squeue when it retrieves
171  * a chain of packets in the poll mode. The packets have gone through the
172  * data link processing but not IP processing. For performance and latency
173  * reasons, the squeue wants to process the chain in line instead of feeding
174  * it back via ip_input path.
175  *
176  * We set up the ip_recv_attr_t with IRAF_TARGET_SQP to that ip_fanout_v4
177  * will pass back any TCP packets matching the target sqp to
178  * ip_input_common_v4 using ira_target_sqp_mp. Other packets are handled by
179  * ip_input_v4 and ip_fanout_v4 as normal.
180  * The TCP packets that match the target squeue are returned to the caller
181  * as a b_next chain after each packet has been prepend with an mblk
182  * from ip_recv_attr_to_mblk.
183  */
184 mblk_t *
185 ip_accept_tcp(ill_t *ill, ill_rx_ring_t *ip_ring, squeue_t *target_sqp,
186     mblk_t *mp_chain, mblk_t **last, uint_t *cnt)
187 {
188         return (ip_input_common_v4(ill, ip_ring, mp_chain, NULL, target_sqp,
189             last, cnt));
190 }

192 /*
193  * Used by ip_input and ip_accept_tcp
```



```
194  * The last three arguments are only used by ip_accept_tcp, and mhip is
195  * only used by ip_input.
196  */
197 mblk_t *
198 ip_input_common_v4(ill_t *ill, ill_rx_ring_t *ip_ring, mblk_t *mp_chain,
199     struct mac_header_info_s *mhip, squeue_t *target_sqp,
200     mblk_t **last, uint_t *cnt)
201 {
202         mblk_t          *mp;
203         ipha_t          *ipha;
204         ip_recv_attr_t  iras;   /* Receive attributes */
205         rtc_t           rtc;
206         iaflags_t       chain_flags = 0;        /* Fixed for chain */
207         mblk_t          *ahead = NULL;  /* Accepted head */
208         mblk_t          *atail = NULL;  /* Accepted tail */
209         uint_t          acnt = 0;       /* Accepted count */

211         ASSERT(mp_chain != NULL);
212         ASSERT(ill != NULL);

214         /* These ones do not change as we loop over packets */
215         iras.ira_ill = iras.ira_rill = ill;
216         iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
217         iras.ira_rifindex = iras.ira_ruifindex;
218         iras.ira_sqp = NULL;
219         iras.ira_ring = ip_ring;
220         /* For ECMP and outbound transmit ring selection */
221         iras.ira_xmit_hint = ILL_RING_TO_XMIT_HINT(ip_ring);

223         iras.ira_target_sqp = target_sqp;
224         iras.ira_target_sqp_mp = NULL;
225         if (target_sqp != NULL)
226                 chain_flags |= IRAF_TARGET_SQP;

228         /*
229          * We try to have a mhip pointer when possible, but
230          * it might be NULL in some cases. In those cases we
231          * have to assume unicast.
232          */
233         iras.ira_mhip = mhip;
234         iras.ira_flags = 0;
235         if (mhip != NULL) {
236                 switch (mhip->mhi_dsttype) {
237                 case MAC_ADDRTYPE_MULTICAST :
238                         chain_flags |= IRAF_L2DST_MULTICAST;
239                         break;
240                 case MAC_ADDRTYPE_BROADCAST :
241                         chain_flags |= IRAF_L2DST_BROADCAST;
242                         break;
243                 }
244         }

246         /*
247          * Initialize the one-element route cache.
248          *
249          * We do ire caching from one iteration to
250          * another. In the event the packet chain contains
251          * all packets from the same dst, this caching saves
252          * an ire_route_recursive for each of the succeeding
253          * packets in a packet chain.
254          */
255         rtc.rtc_ire = NULL;
256         rtc.rtc_ipaddr = INADDR_ANY;

258         /* Loop over b_next */
259         for (mp = mp_chain; mp != NULL; mp = mp_chain) {
```

```
260                 mp_chain = mp->b_next;
261                 mp->b_next = NULL;

263                 ASSERT(DB_TYPE(mp) == M_DATA);


266                 /*
267                  * if db_ref > 1 then copymsg and free original. Packet
268                  * may be changed and we do not want the other entity
269                  * who has a reference to this message to trip over the
270                  * changes. This is a blind change because trying to
271                  * catch all places that might change the packet is too
272                  * difficult.
273                  *
274                  * This corresponds to the fast path case, where we have
275                  * a chain of M_DATA mblks.  We check the db_ref count
276                  * of only the 1st data block in the mblk chain. There
277                  * doesn't seem to be a reason why a device driver would
278                  * send up data with varying db_ref counts in the mblk
279                  * chain. In any case the Fast path is a private
280                  * interface, and our drivers don't do such a thing.
281                  * Given the above assumption, there is no need to walk
282                  * down the entire mblk chain (which could have a
283                  * potential performance problem)
284                  *
285                  * The "(DB_REF(mp) > 1)" check was moved from ip_rput()
286                  * to here because of exclusive ip stacks and vnics.
287                  * Packets transmitted from exclusive stack over vnic
288                  * can have db_ref > 1 and when it gets looped back to
289                  * another vnic in a different zone, you have ip_input()
290                  * getting dblks with db_ref > 1. So if someone
291                  * complains of TCP performance under this scenario,
292                  * take a serious look here on the impact of copymsg().
293                  */
294                 if (DB_REF(mp) > 1) {
295                         if ((mp = ip_fix_dbref(mp, &iras)) == NULL) {
296                                 /* mhip might point into 1st packet in chain */
297                                 iras.ira_mhip = NULL;
298                                 continue;
299                         }
300                 }

302                 /*
303                  * IP header ptr not aligned?
304                  * OR IP header not complete in first mblk
305                  */
306                 ipha = (ipha_t *)mp->b_rptr;
307                 if (!OK_32PTR(ipha) || MBLKL(mp) < IP_SIMPLE_HDR_LENGTH) {
308                         mp = ip_check_and_align_header(mp, IP_SIMPLE_HDR_LENGTH,
309                             &iras);
310                         if (mp == NULL) {
311                                 /* mhip might point into 1st packet in chain */
312                                 iras.ira_mhip = NULL;
313                                 continue;
314                         }
315                         ipha = (ipha_t *)mp->b_rptr;
316                 }

318                 /* Protect against a mix of Ethertypes and IP versions */
319                 if (IPH_HDR_VERSION(ipha) != IPV4_VERSION) {
320                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
321                         ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
322                         freemsg(mp);
323                         /* mhip might point into 1st packet in the chain. */
324                         iras.ira_mhip = NULL;
325                         continue;
```



```
326                 }

328                 /*
329                  * Check for Martian addrs; we have to explicitly
330                  * test for for zero dst since this is also used as
331                  * an indication that the rtc is not used.
332                  */
333                 if (ipha->ipha_dst == INADDR_ANY) {
334                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInAddrErrors);
335                         ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
336                         freemsg(mp);
337                         /* mhip might point into 1st packet in the chain. */
338                         iras.ira_mhip = NULL;
339                         continue;
340                 }

342                 /*
343                  * Keep L2SRC from a previous packet in chain since mhip
344                  * might point into an earlier packet in the chain.
345                  * Keep IRAF_VERIFIED_SRC to avoid redoing broadcast
346                  * source check in forwarding path.
347                  */
348                 chain_flags |= (iras.ira_flags &
349                     (IRAF_L2SRC_SET|IRAF_VERIFIED_SRC));

351                 iras.ira_flags = IRAF_IS_IPV4 | IRAF_VERIFY_IP_CKSUM |
352                     IRAF_VERIFY_ULP_CKSUM | chain_flags;
353                 iras.ira_free_flags = 0;
354                 iras.ira_cred = NULL;
355                 iras.ira_cpid = NOPID;
356                 iras.ira_tsl = NULL;
357                 iras.ira_zoneid = ALL_ZONES;    /* Default for forwarding */

359                 /*
360                  * We must count all incoming packets, even if they end
361                  * up being dropped later on. Defer counting bytes until
362                  * we have the whole IP header in first mblk.
363                  */
364                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInReceives);

366                 iras.ira_pktlen = ntohs(ipha->ipha_length);
367                 UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCInOctets,
368                     iras.ira_pktlen);

370                 /*
371                  * Call one of:
372                  *      ill_input_full_v4
373                  *      ill_input_short_v4
374                  * The former is used in unusual cases. See ill_set_inputfn().
375                  */
376                 (*ill->ill_inputfn)(mp, ipha, &ipha->ipha_dst, &iras, &rtc);

378                 /* Any references to clean up? No hold on ira_ill */
379                 if (iras.ira_flags & (IRAF_IPSEC_SECURE|IRAF_SYSTEM_LABELED))
380                         ira_cleanup(&iras, B_FALSE);

382                 if (iras.ira_target_sqp_mp != NULL) {
383                         /* Better be called from ip_accept_tcp */
384                         ASSERT(target_sqp != NULL);

386                         /* Found one packet to accept */
387                         mp = iras.ira_target_sqp_mp;
388                         iras.ira_target_sqp_mp = NULL;
389                         ASSERT(ip_recv_attr_is_mblk(mp));

391                         if (atail != NULL)
```

```
392                                 atail->b_next = mp;
393                         else
394                                 ahead = mp;
395                         atail = mp;
396                         acnt++;
397                         mp = NULL;
398                 }
399                 /* mhip might point into 1st packet in the chain. */
400                 iras.ira_mhip = NULL;
401         }
402         /* Any remaining references to the route cache? */
403         if (rtc.rtc_ire != NULL) {
404                 ASSERT(rtc.rtc_ipaddr != INADDR_ANY);
405                 ire_refrele(rtc.rtc_ire);
406         }

408         if (ahead != NULL) {
409                 /* Better be called from ip_accept_tcp */
410                 ASSERT(target_sqp != NULL);
411                 *last = atail;
412                 *cnt = acnt;
413                 return (ahead);
414         }

416         return (NULL);
417 }

419 /*
420  * This input function is used when
421  *  - is_system_labeled()
422  *  - CGTP filtering
423  *  - DHCP unicast before we have an IP address configured
424  *  - there is an listener for IPPROTO_RSVP
425  */
426 void
427 ill_input_full_v4(mblk_t *mp, void *iph_arg, void *nexthop_arg,
428     ip_recv_attr_t *ira, rtc_t *rtc)
429 {
430         ipha_t          *ipha = (ipha_t *)iph_arg;
431         ipaddr_t        nexthop = *(ipaddr_t *)nexthop_arg;
432         ill_t           *ill = ira->ira_ill;
433         ip_stack_t      *ipst = ill->ill_ipst;
434         int             cgtp_flt_pkt;

436         ASSERT(ira->ira_tsl == NULL);

438         /*
439          * Attach any necessary label information to
440          * this packet
441          */
442         if (is_system_labeled()) {
443                 ira->ira_flags |= IRAF_SYSTEM_LABELED;

445                 /*
446                  * This updates ira_cred, ira_tsl and ira_free_flags based
447                  * on the label.
448                  */
449                 if (!tsol_get_pkt_label(mp, IPV4_VERSION, ira)) {
450                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
451                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
452                         freemsg(mp);
453                         return;
454                 }
455                 /* Note that ira_tsl can be NULL here. */

457                 /* tsol_get_pkt_label sometimes does pullupmsg */
```



```
458                 ipha = (ipha_t *)mp->b_rptr;
459         }

461         /*
462          * Invoke the CGTP (multirouting) filtering module to process
463          * the incoming packet. Packets identified as duplicates
464          * must be discarded. Filtering is active only if the
465          * the ip_cgtp_filter ndd variable is non-zero.
466          */
467         cgtp_flt_pkt = CGTP_IP_PKT_NOT_CGTP;
468         if (ipst->ips_ip_cgtp_filter &&
469             ipst->ips_ip_cgtp_filter_ops != NULL) {
470                 netstackid_t stackid;

472                 stackid = ipst->ips_netstack->netstack_stackid;
473                 /*
474                  * CGTP and IPMP are mutually exclusive so
475                  * phyint_ifindex is fine here.
476                  */
477                 cgtp_flt_pkt =
478                     ipst->ips_ip_cgtp_filter_ops->cfo_filter(stackid,
479                     ill->ill_phyint->phyint_ifindex, mp);
480                 if (cgtp_flt_pkt == CGTP_IP_PKT_DUPLICATE) {
481                         ip_drop_input("CGTP_IP_PKT_DUPLICATE", mp, ill);
482                         freemsg(mp);
483                         return;
484                 }
485         }

487         /*
488          * Brutal hack for DHCPv4 unicast: RFC2131 allows a DHCP
489          * server to unicast DHCP packets to a DHCP client using the
490          * IP address it is offering to the client.  This can be
491          * disabled through the "broadcast bit", but not all DHCP
492          * servers honor that bit.  Therefore, to interoperate with as
493          * many DHCP servers as possible, the DHCP client allows the
494          * server to unicast, but we treat those packets as broadcast
495          * here.  Note that we don't rewrite the packet itself since
496          * (a) that would mess up the checksums and (b) the DHCP
497          * client conn is bound to INADDR_ANY so ip_fanout_udp() will
498          * hand it the packet regardless.
499          */
500         if (ill->ill_dhcpinit != 0 &&
501             ipha->ipha_version_and_hdr_length == IP_SIMPLE_HDR_VERSION &&
502             ipha->ipha_protocol == IPPROTO_UDP) {
503                 udpha_t *udpha;

505                 ipha = ip_pullup(mp, sizeof (ipha_t) + sizeof (udpha_t), ira);
506                 if (ipha == NULL) {
507                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
508                         ip_drop_input("ipIfStatsInDiscards - dhcp", mp, ill);
509                         freemsg(mp);
510                         return;
511                 }
512                 /* Reload since pullupmsg() can change b_rptr. */
513                 udpha = (udpha_t *)&ipha[1];

515                 if (ntohs(udpha->uha_dst_port) == IPPORT_BOOTPC) {
516                         DTRACE_PROBE2(ip4__dhcpinit__pkt, ill_t *, ill,
517                             mblk_t *, mp);
518                         /*
519                          * This assumes that we deliver to all conns for
520                          * multicast and broadcast packets.
521                          */
522                         nexthop = INADDR_BROADCAST;
523                         ira->ira_flags |= IRAF_DHCP_UNICAST;
```

```
524                 }
525         }

527         /*
528          * If rsvpd is running, let RSVP daemon handle its processing
529          * and forwarding of RSVP multicast/unicast packets.
530          * If rsvpd is not running but mrouted is running, RSVP
531          * multicast packets are forwarded as multicast traffic
532          * and RSVP unicast packets are forwarded by unicast router.
533          * If neither rsvpd nor mrouted is running, RSVP multicast
534          * packets are not forwarded, but the unicast packets are
535          * forwarded like unicast traffic.
536          */
537         if (ipha->ipha_protocol == IPPROTO_RSVP &&
538             ipst->ips_ipcl_proto_fanout_v4[IPPROTO_RSVP].connf_head != NULL) {
539                 /* RSVP packet and rsvpd running. Treat as ours */
540                 ip2dbg(("ip_input: RSVP for us: 0x%x\n", ntohl(nexthop)));
541                 /*
542                  * We use a multicast address to get the packet to
543                  * ire_recv_multicast_v4. There will not be a membership
544                  * check since we set IRAF_RSVP
545                  */
546                 nexthop = htonl(INADDR_UNSPEC_GROUP);
547                 ira->ira_flags |= IRAF_RSVP;
548         }

550         ill_input_short_v4(mp, ipha, &nexthop, ira, rtc);
551 }

553 /*
554  * This is the tail-end of the full receive side packet handling.
555  * It can be used directly when the configuration is simple.
556  */
557 void
558 ill_input_short_v4(mblk_t *mp, void *iph_arg, void *nexthop_arg,
559     ip_recv_attr_t *ira, rtc_t *rtc)
560 {
561         ire_t           *ire;
562         uint_t          opt_len;
563         ill_t           *ill = ira->ira_ill;
564         ip_stack_t      *ipst = ill->ill_ipst;
565         uint_t          pkt_len;
566         ssize_t         len;
567         ipha_t          *ipha = (ipha_t *)iph_arg;
568         ipaddr_t        nexthop = *(ipaddr_t *)nexthop_arg;
569         ilb_stack_t     *ilbs = ipst->ips_netstack->netstack_ilb;
570         uint_t          irr_flags;
571 #define rptr    ((uchar_t *)ipha)

573         ASSERT(DB_TYPE(mp) == M_DATA);

575         /*
576          * The following test for loopback is faster than
577          * IP_LOOPBACK_ADDR(), because it avoids any bitwise
578          * operations.
579          * Note that these addresses are always in network byte order
580          */
581         if (((*(uchar_t *)&ipha->ipha_dst) == IN_LOOPBACKNET) ||
582             ((*(uchar_t *)&ipha->ipha_src) == IN_LOOPBACKNET)) {
583                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInAddrErrors);
584                 ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
585                 freemsg(mp);
586                 return;
587         }

589         len = mp->b_wptr - rptr;
```



```
590         pkt_len = ira->ira_pktlen;

592         /* multiple mblk or too short */
593         len -= pkt_len;
594         if (len != 0) {
595                 mp = ip_check_length(mp, rptr, len, pkt_len,
596                     IP_SIMPLE_HDR_LENGTH, ira);
597                 if (mp == NULL)
598                         return;
599                 ipha = (ipha_t *)mp->b_rptr;
600         }

602         DTRACE_IP7(receive, mblk_t *, mp, conn_t *, NULL, void_ip_t *,
603             ipha, __dtrace_ipsr_ill_t *, ill, ipha_t *, ipha, ip6_t *, NULL,
604             int, 0);

606         /*
607          * The event for packets being received from a 'physical'
608          * interface is placed after validation of the source and/or
609          * destination address as being local so that packets can be
610          * redirected to loopback addresses using ipnat.
611          */
612         DTRACE_PROBE4(ip4__physical__in__start,
613             ill_t *, ill, ill_t *, NULL,
614             ipha_t *, ipha, mblk_t *, mp);

616         if (HOOKS4_INTERESTED_PHYSICAL_IN(ipst)) {
617                 int     ll_multicast = 0;
618                 int     error;
619                 ipaddr_t orig_dst = ipha->ipha_dst;

621                 if (ira->ira_flags & IRAF_L2DST_MULTICAST)
622                         ll_multicast = HPE_MULTICAST;
623                 else if (ira->ira_flags & IRAF_L2DST_BROADCAST)
624                         ll_multicast = HPE_BROADCAST;

626                 FW_HOOKS(ipst->ips_ip4_physical_in_event,
627                     ipst->ips_ipv4firewall_physical_in,
628                     ill, NULL, ipha, mp, mp, ll_multicast, ipst, error);

630                 DTRACE_PROBE1(ip4__physical__in__end, mblk_t *, mp);

632                 if (mp == NULL)
633                         return;
634                 /* The length could have changed */
635                 ipha = (ipha_t *)mp->b_rptr;
636                 ira->ira_pktlen = ntohs(ipha->ipha_length);
637                 pkt_len = ira->ira_pktlen;

639                 /*
640                  * In case the destination changed we override any previous
641                  * change to nexthop.
642                  */
643                 if (orig_dst != ipha->ipha_dst)
644                         nexthop = ipha->ipha_dst;
645                 if (nexthop == INADDR_ANY) {
646                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInAddrErrors);
647                         ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
648                         freemsg(mp);
649                         return;
650                 }
651         }

653         if (ipst->ips_ip4_observe.he_interested) {
654                 zoneid_t dzone;
```

```
656                 /*
657                  * On the inbound path the src zone will be unknown as
658                  * this packet has come from the wire.
659                  */
660                 dzone = ip_get_zoneid_v4(nexthop, mp, ira, ALL_ZONES);
661                 ipobs_hook(mp, IPOBS_HOOK_INBOUND, ALL_ZONES, dzone, ill, ipst);
662         }

664         /*
665          * If there is a good HW IP header checksum we clear the need
666          * look at the IP header checksum.
667          */
668         if ((DB_CKSUMFLAGS(mp) & HCK_IPV4_HDRCKSUM) &&
669             ILL_HCKSUM_CAPABLE(ill) && dohwcksum) {
670                 /* Header checksum was ok. Clear the flag */
671                 DB_CKSUMFLAGS(mp) &= ~HCK_IPV4_HDRCKSUM;
672                 ira->ira_flags &= ~IRAF_VERIFY_IP_CKSUM;
673         }

675         /*
676          * Here we check to see if we machine is setup as
677          * L3 loadbalancer and if the incoming packet is for a VIP
678          *
679          * Check the following:
680          * - there is at least a rule
681          * - protocol of the packet is supported
682          */
683         if (ilb_has_rules(ilbs) && ILB_SUPP_L4(ipha->ipha_protocol)) {
684                 ipaddr_t        lb_dst;
685                 int             lb_ret;

687                 /* For convenience, we pull up the mblk. */
688                 if (mp->b_cont != NULL) {
689                         if (pullupmsg(mp, -1) == 0) {
690                                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
691                                 ip_drop_input("ipIfStatsInDiscards - pullupmsg",
692                                     mp, ill);
693                                 freemsg(mp);
694                                 return;
695                         }
696                         ipha = (ipha_t *)mp->b_rptr;
697                 }

699                 /*
700                  * We just drop all fragments going to any VIP, at
701                  * least for now....
702                  */
703                 if (ntohs(ipha->ipha_fragment_offset_and_flags) &
704                     (IPH_MF | IPH_OFFSET)) {
705                         if (!ilb_rule_match_vip_v4(ilbs, nexthop, NULL)) {
706                                 goto after_ilb;
707                         }

709                         ILB_KSTAT_UPDATE(ilbs, ip_frag_in, 1);
710                         ILB_KSTAT_UPDATE(ilbs, ip_frag_dropped, 1);
711                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
712                         ip_drop_input("ILB fragment", mp, ill);
713                         freemsg(mp);
714                         return;
715                 }
716                 lb_ret = ilb_check_v4(ilbs, ill, mp, ipha, ipha->ipha_protocol,
717                     (uint8_t *)ipha + IPH_HDR_LENGTH(ipha), &lb_dst);

719                 if (lb_ret == ILB_DROPPED) {
720                         /* Is this the right counter to increase? */
721                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
```



```
722                         ip_drop_input("ILB_DROPPED", mp, ill);
723                         freemsg(mp);
724                         return;
725                 }
726                 if (lb_ret == ILB_BALANCED) {
727                         /* Set the dst to that of the chosen server */
728                         nexthop = lb_dst;
729                         DB_CKSUMFLAGS(mp) = 0;
730                 }
731         }

733 after_ilb:
734         opt_len = ipha->ipha_version_and_hdr_length - IP_SIMPLE_HDR_VERSION;
735         ira->ira_ip_hdr_length = IP_SIMPLE_HDR_LENGTH;
736         if (opt_len != 0) {
737                 int error = 0;

739                 ira->ira_ip_hdr_length += (opt_len << 2);
740                 ira->ira_flags |= IRAF_IPV4_OPTIONS;

742                 /* IP Options present!  Validate the length. */
743                 mp = ip_check_optlen(mp, ipha, opt_len, pkt_len, ira);
744                 if (mp == NULL)
745                         return;

747                 /* Might have changed */
748                 ipha = (ipha_t *)mp->b_rptr;

750                 /* Verify IP header checksum before parsing the options */
751                 if ((ira->ira_flags & IRAF_VERIFY_IP_CKSUM) &&
752                     ip_csum_hdr(ipha)) {
753                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
754                         ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
755                         freemsg(mp);
756                         return;
757                 }
758                 ira->ira_flags &= ~IRAF_VERIFY_IP_CKSUM;

760                 /*
761                  * Go off to ip_input_options which returns the next hop
762                  * destination address, which may have been affected
763                  * by source routing.
764                  */
765                 IP_STAT(ipst, ip_opt);

767                 nexthop = ip_input_options(ipha, nexthop, mp, ira, &error);
768                 if (error != 0) {
769                         /*
770                          * An ICMP error has been sent and the packet has
771                          * been dropped.
772                          */
773                         return;
774                 }
775         }

777         if (ill->ill_flags & ILLF_ROUTER)
778                 irr_flags = IRR_ALLOCATE;
779         else
780                 irr_flags = IRR_NONE;

782         /* Can not use route cache with TX since the labels can differ */
783         if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
784                 if (CLASSD(nexthop)) {
785                         ire = ire_multicast(ill);
786                 } else {
787                         /* Match destination and label */
```

```
788                         ire = ire_route_recursive_v4(nexthop, 0, NULL,
789                             ALL_ZONES, ira->ira_tsl, MATCH_IRE_SECATTR,
790                             irr_flags, ira->ira_xmit_hint, ipst, NULL, NULL,
791                             NULL);
792                 }
793                 /* Update the route cache so we do the ire_refrele */
794                 ASSERT(ire != NULL);
795                 if (rtc->rtc_ire != NULL)
796                         ire_refrele(rtc->rtc_ire);
797                 rtc->rtc_ire = ire;
798                 rtc->rtc_ipaddr = nexthop;
799         } else if (nexthop == rtc->rtc_ipaddr && rtc->rtc_ire != NULL) {
800                 /* Use the route cache */
801                 ire = rtc->rtc_ire;
802         } else {
803                 /* Update the route cache */
804                 if (CLASSD(nexthop)) {
805                         ire = ire_multicast(ill);
806                 } else {
807                         /* Just match the destination */
808                         ire = ire_route_recursive_dstonly_v4(nexthop, irr_flags,
809                             ira->ira_xmit_hint, ipst);
810                 }
811                 ASSERT(ire != NULL);
812                 if (rtc->rtc_ire != NULL)
813                         ire_refrele(rtc->rtc_ire);
814                 rtc->rtc_ire = ire;
815                 rtc->rtc_ipaddr = nexthop;
816         }

818         ire->ire_ib_pkt_count++;

820         /*
821          * Based on ire_type and ire_flags call one of:
822          *      ire_recv_local_v4 - for IRE_LOCAL
823          *      ire_recv_loopback_v4 - for IRE_LOOPBACK
824          *      ire_recv_multirt_v4 - if RTF_MULTIRT
825          *      ire_recv_noroute_v4 - if RTF_REJECT or RTF_BLACHOLE
826          *      ire_recv_multicast_v4 - for IRE_MULTICAST
827          *      ire_recv_broadcast_v4 - for IRE_BROADCAST
828          *      ire_recv_noaccept_v4 - for ire_noaccept ones
829          *      ire_recv_forward_v4 - for the rest.
830          */
831         (*ire->ire_recvfn)(ire, mp, ipha, ira);
832 }
833 #undef rptr

835 /*
836  * ire_recvfn for IREs that need forwarding
837  */
838 void
839 ire_recv_forward_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_recv_attr_t *ira)
840 {
841         ipha_t          *ipha = (ipha_t *)iph_arg;
842         ill_t           *ill = ira->ira_ill;
843         ip_stack_t      *ipst = ill->ill_ipst;
844         ill_t           *dst_ill;
845         nce_t           *nce;
846         ipaddr_t        src = ipha->ipha_src;
847         uint32_t        added_tx_len;
848         uint32_t        mtu, iremtu;

850         if (ira->ira_flags & (IRAF_L2DST_MULTICAST|IRAF_L2DST_BROADCAST)) {
851                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
852                 ip_drop_input("l2 multicast not forwarded", mp, ill);
853                 freemsg(mp);
```



```
854                 return;
855         }

857         if (!(ill->ill_flags & ILLF_ROUTER) && !ip_source_routed(ipha, ipst)) {
858                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
859                 ip_drop_input("ipIfStatsForwProhibits", mp, ill);
860                 freemsg(mp);
861                 return;
862         }

864         /*
865          * Either ire_nce_capable or ire_dep_parent would be set for the IRE
866          * when it is found by ire_route_recursive, but that some other thread
867          * could have changed the routes with the effect of clearing
868          * ire_dep_parent. In that case we'd end up dropping the packet, or
869          * finding a new nce below.
870          * Get, allocate, or update the nce.
871          * We get a refhold on ire_nce_cache as a result of this to avoid races
872          * where ire_nce_cache is deleted.
873          *
874          * This ensures that we don't forward if the interface is down since
875          * ipif_down removes all the nces.
876          */
877         mutex_enter(&ire->ire_lock);
878         nce = ire->ire_nce_cache;
879         if (nce == NULL) {
880                 /* Not yet set up - try to set one up */
881                 mutex_exit(&ire->ire_lock);
882                 (void) ire_revalidate_nce(ire);
883                 mutex_enter(&ire->ire_lock);
884                 nce = ire->ire_nce_cache;
885                 if (nce == NULL) {
886                         mutex_exit(&ire->ire_lock);
887                         /* The ire_dep_parent chain went bad, or no memory */
888                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
889                         ip_drop_input("No ire_dep_parent", mp, ill);
890                         freemsg(mp);
891                         return;
892                 }
893         }
894         nce_refhold(nce);
895         mutex_exit(&ire->ire_lock);

897         if (nce->nce_is_condemned) {
898                 nce_t *nce1;

900                 nce1 = ire_handle_condemned_nce(nce, ire, ipha, NULL, B_FALSE);
901                 nce_refrele(nce);
902                 if (nce1 == NULL) {
903                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
904                         ip_drop_input("No nce", mp, ill);
905                         freemsg(mp);
906                         return;
907                 }
908                 nce = nce1;
909         }
910         dst_ill = nce->nce_ill;

912         /*
913          * Unless we are forwarding, drop the packet.
914          * We have to let source routed packets through if they go out
915          * the same interface i.e., they are 'ping -l' packets.
916          */
917         if (!(dst_ill->ill_flags & ILLF_ROUTER) &&
918             !(ip_source_routed(ipha, ipst) && dst_ill == ill)) {
919                 if (ip_source_routed(ipha, ipst)) {
```

```
920                     ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
921                     icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
922                     nce_refrele(nce);
923                     return;
924             }
925             BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
926             ip_drop_input("ipIfStatsForwProhibits", mp, ill);
927             freemsg(mp);
928             nce_refrele(nce);
929             return;
930     }

932     if (ire->ire_zoneid != GLOBAL_ZONEID && ire->ire_zoneid != ALL_ZONES) {
933             ipaddr_t        dst = ipha->ipha_dst;

935             ire->ire_ib_pkt_count--;
936             /*
937              * Should only use IREs that are visible from the
938              * global zone for forwarding.
939              * Take a source route into account the same way as ip_input
940              * did.
941              */
942             if (ira->ira_flags & IRAF_IPV4_OPTIONS) {
943                     int             error = 0;

945                     dst = ip_input_options(ipha, dst, mp, ira, &error);
946                     ASSERT(error == 0);     /* ip_input checked */
947             }
948             ire = ire_route_recursive_v4(dst, 0, NULL, GLOBAL_ZONEID,
949                 ira->ira_tsl, MATCH_IRE_SECATTR,
950                 (ill->ill_flags & ILLF_ROUTER) ? IRR_ALLOCATE : IRR_NONE,
951                 ira->ira_xmit_hint, ipst, NULL, NULL, NULL);
952             ire->ire_ib_pkt_count++;
953             (*ire->ire_recvfn)(ire, mp, ipha, ira);
954             ire_refrele(ire);
955             nce_refrele(nce);
956             return;
957     }

959     /*
960      * ipIfStatsHCInForwDatagrams should only be increment if there
961      * will be an attempt to forward the packet, which is why we
962      * increment after the above condition has been checked.
963      */
964     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInForwDatagrams);

966     /* Initiate Read side IPPF processing */
967     if (IPP_ENABLED(IPP_FWD_IN, ipst)) {
968             /* ip_process translates an IS_UNDER_IPMP */
969             mp = ip_process(IPP_FWD_IN, mp, ill, ill);
970             if (mp == NULL) {
971                     /* ip_drop_packet and MIB done */
972                     ip2dbg(("ire_recv_forward_v4: pkt dropped/deferred "
973                         "during IPPF processing\n"));
974                     nce_refrele(nce);
975                     return;
976             }
977     }

979     DTRACE_PROBE4(ip4__forwarding__start,
980         ill_t *, ill, ill_t *, dst_ill, ipha_t *, ipha, mblk_t *, mp);

982     if (HOOKS4_INTERESTED_FORWARDING(ipst)) {
983             int error;

985             FW_HOOKS(ipst->ips_ip4_forwarding_event,
```



```
986                 ipst->ips_ipv4firewall_forwarding,
987                 ill, dst_ill, ipha, mp, mp, 0, ipst, error);

989             DTRACE_PROBE1(ip4__forwarding__end, mblk_t *, mp);

991             if (mp == NULL) {
992                     nce_refrele(nce);
993                     return;
994             }
995             /*
996              * Even if the destination was changed by the filter we use the
997              * forwarding decision that was made based on the address
998              * in ip_input.
999              */

1001            /* Might have changed */
1002            ipha = (ipha_t *)mp->b_rptr;
1003            ira->ira_pktlen = ntohs(ipha->ipha_length);
1004    }

1006    /* Packet is being forwarded. Turning off hwcksum flag. */
1007    DB_CKSUMFLAGS(mp) = 0;

1009    /*
1010     * Martian Address Filtering [RFC 1812, Section 5.3.7]
1011     * The loopback address check for both src and dst has already
1012     * been checked in ip_input
1013     * In the future one can envision adding RPF checks using number 3.
1014     * If we already checked the same source address we can skip this.
1015     */
1016    if (!(ira->ira_flags & IRAF_VERIFIED_SRC) ||
1017        src != ira->ira_verified_src) {
1018            switch (ipst->ips_src_check) {
1019            case 0:
1020                    break;
1021            case 2:
1022                    if (ip_type_v4(src, ipst) == IRE_BROADCAST) {
1023                            BUMP_MIB(ill->ill_ip_mib,
1024                                ipIfStatsForwProhibits);
1025                            BUMP_MIB(ill->ill_ip_mib,
1026                                ipIfStatsInAddrErrors);
1027                            ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
1028                            freemsg(mp);
1029                            nce_refrele(nce);
1030                            return;
1031                    }
1032                    /* FALLTHRU */

1034            case 1:
1035                    if (CLASSD(src)) {
1036                            BUMP_MIB(ill->ill_ip_mib,
1037                                ipIfStatsForwProhibits);
1038                            BUMP_MIB(ill->ill_ip_mib,
1039                                ipIfStatsInAddrErrors);
1040                            ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
1041                            freemsg(mp);
1042                            nce_refrele(nce);
1043                            return;
1044                    }
1045                    break;
1046            }
1047            /* Remember for next packet */
1048            ira->ira_flags |= IRAF_VERIFIED_SRC;
1049            ira->ira_verified_src = src;
1050    }
```

```
1052         /*
1053          * Check if packet is going out the same link on which it arrived.
1054          * Means we might need to send a redirect.
1055          */
1056         if (IS_ON_SAME_LAN(dst_ill, ill) && ipst->ips_ip_g_send_redirects) {
1057                 ip_send_potential_redirect_v4(mp, ipha, ire, ira);
1058         }

1060         added_tx_len = 0;
1061         if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
1062                 mblk_t          *mp1;
1063                 uint32_t        old_pkt_len = ira->ira_pktlen;

1065                 /* Verify IP header checksum before adding/removing options */
1066                 if ((ira->ira_flags & IRAF_VERIFY_IP_CKSUM) &&
1067                     ip_csum_hdr(ipha)) {
1068                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1069                         ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
1070                         freemsg(mp);
1071                         nce_refrele(nce);
1072                         return;
1073                 }
1074                 ira->ira_flags &= ~IRAF_VERIFY_IP_CKSUM;

1076                 /*
1077                  * Check if it can be forwarded and add/remove
1078                  * CIPSO options as needed.
1079                  */
1080                 if ((mp1 = tsol_ip_forward(ire, mp, ira)) == NULL) {
1081                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
1082                         ip_drop_input("tsol_ip_forward", mp, ill);
1083                         freemsg(mp);
1084                         nce_refrele(nce);
1085                         return;
1086                 }
1087                 /*
1088                  * Size may have changed. Remember amount added in case
1089                  * IP needs to send an ICMP too big.
1090                  */
1091                 mp = mp1;
1092                 ipha = (ipha_t *)mp->b_rptr;
1093                 ira->ira_pktlen = ntohs(ipha->ipha_length);
1094                 ira->ira_ip_hdr_length = IPH_HDR_LENGTH(ipha);
1095                 if (ira->ira_pktlen > old_pkt_len)
1096                         added_tx_len = ira->ira_pktlen - old_pkt_len;

1098                 /* Options can have been added or removed */
1099                 if (ira->ira_ip_hdr_length != IP_SIMPLE_HDR_LENGTH)
1100                         ira->ira_flags |= IRAF_IPV4_OPTIONS;
1101                 else
1102                         ira->ira_flags &= ~IRAF_IPV4_OPTIONS;
1103         }

1105         mtu = dst_ill->ill_mtu;
1106         if ((iremtu = ire->ire_metrics.iulp_mtu) != 0 && iremtu < mtu)
1107                 mtu = iremtu;
1108         ip_forward_xmit_v4(nce, ill, mp, ipha, ira, mtu, added_tx_len);
1109         nce_refrele(nce);
1110 }

1112 /*
1113  * Used for sending out unicast and multicast packets that are
1114  * forwarded.
1115  */
1116 void
1117 ip_forward_xmit_v4(nce_t *nce, ill_t *ill, mblk_t *mp, ipha_t *ipha,
```



```
1118     ip_recv_attr_t *ira, uint32_t mtu, uint32_t added_tx_len)
1119 {
1120         ill_t           *dst_ill = nce->nce_ill;
1121         uint32_t        pkt_len;
1122         uint32_t        sum;
1123         iaflags_t       iraflags = ira->ira_flags;
1124         ip_stack_t      *ipst = ill->ill_ipst;
1125         iaflags_t       ixaflags;

1127         if (ipha->ipha_ttl <= 1) {
1128                 /* Perhaps the checksum was bad */
1129                 if ((iraflags & IRAF_VERIFY_IP_CKSUM) && ip_csum_hdr(ipha)) {
1130                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1131                         ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
1132                         freemsg(mp);
1133                         return;
1134                 }
1135                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1136                 ip_drop_input("ICMP_TTL_EXCEEDED", mp, ill);
1137                 icmp_time_exceeded(mp, ICMP_TTL_EXCEEDED, ira);
1138                 return;
1139         }
1140         ipha->ipha_ttl--;
1141         /* Adjust the checksum to reflect the ttl decrement. */
1142         sum = (int)ipha->ipha_hdr_checksum + IP_HDR_CSUM_TTL_ADJUST;
1143         ipha->ipha_hdr_checksum = (uint16_t)(sum + (sum >> 16));

1145         /* Check if there are options to update */
1146         if (iraflags & IRAF_IPV4_OPTIONS) {
1147                 ASSERT(ipha->ipha_version_and_hdr_length !=
1148                     IP_SIMPLE_HDR_VERSION);
1149                 ASSERT(!(iraflags & IRAF_VERIFY_IP_CKSUM));

1151                 if (!ip_forward_options(mp, ipha, dst_ill, ira)) {
1152                         /* ipIfStatsForwProhibits and ip_drop_input done */
1153                         return;
1154                 }

1156                 ipha->ipha_hdr_checksum = 0;
1157                 ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
1158         }

1160         /* Initiate Write side IPPF processing before any fragmentation */
1161         if (IPP_ENABLED(IPP_FWD_OUT, ipst)) {
1162                 /* ip_process translates an IS_UNDER_IPMP */
1163                 mp = ip_process(IPP_FWD_OUT, mp, dst_ill, dst_ill);
1164                 if (mp == NULL) {
1165                         /* ip_drop_packet and MIB done */
1166                         ip2dbg(("ire_recv_forward_v4: pkt dropped/deferred" \
1167                             " during IPPF processing\n"));
1168                         return;
1169                 }
1170         }

1172         pkt_len = ira->ira_pktlen;

1174         BUMP_MIB(dst_ill->ill_ip_mib, ipIfStatsHCOutForwDatagrams);

1176         ixaflags = IXAF_IS_IPV4 | IXAF_NO_DEV_FLOW_CTL;

1178         if (pkt_len > mtu) {
1179                 /*
1180                  * It needs fragging on its way out.  If we haven't
1181                  * verified the header checksum yet we do it now since
1182                  * are going to put a surely good checksum in the
1183                  * outgoing header, we have to make sure that it
```

```
1184                 * was good coming in.
1185                 */
1186                if ((iraflags & IRAF_VERIFY_IP_CKSUM) && ip_csum_hdr(ipha)) {
1187                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1188                        ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
1189                        freemsg(mp);
1190                        return;
1191                }
1192                if (ipha->ipha_fragment_offset_and_flags & IPH_DF_HTONS) {
1193                        BUMP_MIB(dst_ill->ill_ip_mib, ipIfStatsOutFragFails);
1194                        ip_drop_output("ipIfStatsOutFragFails", mp, dst_ill);
1195                        if (iraflags & IRAF_SYSTEM_LABELED) {
1196                                /*
1197                                 * Remove any CIPSO option added by
1198                                 * tsol_ip_forward, and make sure we report
1199                                 * a path MTU so that there
1200                                 * is room to add such a CIPSO option for future
1201                                 * packets.
1202                                 */
1203                                mtu = tsol_pmtu_adjust(mp, mtu, added_tx_len,
1204                                    AF_INET);
1205                        }
1206
1207                        icmp_frag_needed(mp, mtu, ira);
1208                        return;
1209                }
1210
1211                (void) ip_fragment_v4(mp, nce, ixaflags, pkt_len, mtu,
1212                    ira->ira_xmit_hint, GLOBAL_ZONEID, 0, ip_xmit, NULL);
1213                return;
1214        }
1215
1216        ASSERT(pkt_len == ntohs(((ipha_t *)mp->b_rptr)->ipha_length));
1217        if (iraflags & IRAF_LOOPBACK_COPY) {
1218                /*
1219                 * IXAF_NO_LOOP_ZONEID is not set hence 7th arg
1220                 * is don't care
1221                 */
1222                (void) ip_postfrag_loopcheck(mp, nce,
1223                    ixaflags | IXAF_LOOPBACK_COPY,
1224                    pkt_len, ira->ira_xmit_hint, GLOBAL_ZONEID, 0, NULL);
1225        } else {
1226                (void) ip_xmit(mp, nce, ixaflags, pkt_len, ira->ira_xmit_hint,
1227                    GLOBAL_ZONEID, 0, NULL);
1228        }
1229 }
1230
1231 /*
1232  * ire_recvfn for RTF_REJECT and RTF_BLACKHOLE routes, including IRE_NOROUTE,
1233  * which is what ire_route_recursive returns when there is no matching ire.
1234  * Send ICMP unreachable unless blackhole.
1235  */
1236 void
1237 ire_recv_noroute_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_recv_attr_t *ira)
1238 {
1239        ipha_t          *ipha = (ipha_t *)iph_arg;
1240        ill_t           *ill = ira->ira_ill;
1241        ip_stack_t      *ipst = ill->ill_ipst;
1242
1243        /* Would we have forwarded this packet if we had a route? */
1244        if (ira->ira_flags & (IRAF_L2DST_MULTICAST|IRAF_L2DST_BROADCAST)) {
1245                BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
1246                ip_drop_input("l2 multicast not forwarded", mp, ill);
1247                freemsg(mp);
1248                return;
1249        }
```



```
1251        if (!(ill->ill_flags & ILLF_ROUTER)) {
1252                BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
1253                ip_drop_input("ipIfStatsForwProhibits", mp, ill);
1254                freemsg(mp);
1255                return;
1256        }
1257        /*
1258         * If we had a route this could have been forwarded. Count as such.
1259         *
1260         * ipIfStatsHCInForwDatagrams should only be increment if there
1261         * will be an attempt to forward the packet, which is why we
1262         * increment after the above condition has been checked.
1263         */
1264        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInForwDatagrams);
1265
1266        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInNoRoutes);
1267
1268        ip_rts_change(RTM_MISS, ipha->ipha_dst, 0, 0, 0, 0, 0, 0, RTA_DST,
1269            ipst);
1270
1271        if (ire->ire_flags & RTF_BLACKHOLE) {
1272                ip_drop_input("ipIfStatsInNoRoutes RTF_BLACKHOLE", mp, ill);
1273                freemsg(mp);
1274        } else {
1275                ip_drop_input("ipIfStatsInNoRoutes RTF_REJECT", mp, ill);
1276
1277                if (ip_source_routed(ipha, ipst)) {
1278                        icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
1279                } else {
1280                        icmp_unreachable(mp, ICMP_HOST_UNREACHABLE, ira);
1281                }
1282        }
1283 }
1284
1285 /*
1286  * ire_recvfn for IRE_LOCALs marked with ire_noaccept. Such IREs are used for
1287  * VRRP when in noaccept mode.
1288  * We silently drop the packet. ARP handles packets even if noaccept is set.
1289  */
1290 /* ARGSUSED */
1291 void
1292 ire_recv_noaccept_v4(ire_t *ire, mblk_t *mp, void *iph_arg,
1293     ip_recv_attr_t *ira)
1294 {
1295        ill_t           *ill = ira->ira_ill;
1296
1297        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1298        ip_drop_input("ipIfStatsInDiscards - noaccept", mp, ill);
1299        freemsg(mp);
1300 }
1301
1302 /*
1303  * ire_recvfn for IRE_BROADCAST.
1304  */
1305 void
1306 ire_recv_broadcast_v4(ire_t *ire, mblk_t *mp, void *iph_arg,
1307     ip_recv_attr_t *ira)
1308 {
1309        ipha_t          *ipha = (ipha_t *)iph_arg;
1310        ill_t           *ill = ira->ira_ill;
1311        ill_t           *dst_ill = ire->ire_ill;
1312        ip_stack_t      *ipst = ill->ill_ipst;
1313        ire_t           *alt_ire;
1314        nce_t           *nce;
1315        ipaddr_t        ipha_dst;
```

```
1317         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInBcastPkts);

1319         /* Tag for higher-level protocols */
1320         ira->ira_flags |= IRAF_BROADCAST;

1322         /*
1323          * Whether local or directed broadcast forwarding: don't allow
1324          * for TCP.
1325          */
1326         if (ipha->ipha_protocol == IPPROTO_TCP) {
1327                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1328                 ip_drop_input("ipIfStatsInDiscards", mp, ill);
1329                 freemsg(mp);
1330                 return;
1331         }

1333         /*
1334          * So that we don't end up with dups, only one ill an IPMP group is
1335          * nominated to receive broadcast traffic.
1336          * If we have no cast_ill we are liberal and accept everything.
1337          */
1338         if (IS_UNDER_IPMP(ill)) {
1339                 /* For an under ill_grp can change under lock */
1340                 rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1341                 if (!ill->ill_nom_cast && ill->ill_grp != NULL &&
1342                     ill->ill_grp->ig_cast_ill != NULL) {
1343                         rw_exit(&ipst->ips_ill_g_lock);
1344                         /* No MIB since this is normal operation */
1345                         ip_drop_input("not nom_cast", mp, ill);
1346                         freemsg(mp);
1347                         return;
1348                 }
1349                 rw_exit(&ipst->ips_ill_g_lock);

1351                 ira->ira_ruifindex = ill_get_upper_ifindex(ill);
1352         }

1354         /*
1355          * After reassembly and IPsec we will need to duplicate the
1356          * broadcast packet for all matching zones on the ill.
1357          */
1358         ira->ira_zoneid = ALL_ZONES;

1360         /*
1361          * Check for directed broadcast i.e. ire->ire_ill is different than
1362          * the incoming ill.
1363          * The same broadcast address can be assigned to multiple interfaces
1364          * so have to check explicitly for that case by looking up the alt_ire
1365          */
1366         if (dst_ill == ill && !(ire->ire_flags & RTF_MULTIRT)) {
1367                 /* Reassemble on the ill on which the packet arrived */
1368                 ip_input_local_v4(ire, mp, ipha, ira);
1369                 /* Restore */
1370                 ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;
1371                 return;
1372         }

1374         /* Is there an IRE_BROADCAST on the incoming ill? */
1375         ipha_dst = ((ira->ira_flags & IRAF_DHCP_UNICAST) ? INADDR_BROADCAST :
1376             ipha->ipha_dst);
1377         alt_ire = ire_ftable_lookup_v4(ipha_dst, 0, 0, IRE_BROADCAST, ill,
1378             ALL_ZONES, ira->ira_tsl,
1379             MATCH_IRE_TYPE|MATCH_IRE_ILL|MATCH_IRE_SECATTR, 0, ipst, NULL);
1380         if (alt_ire != NULL) {
1381                 /* Not a directed broadcast */
```



```
1382                 /*
1383                  * In the special case of multirouted broadcast
1384                  * packets, we unconditionally need to "gateway"
1385                  * them to the appropriate interface here so that reassembly
1386                  * works. We know that the IRE_BROADCAST on cgtp0 doesn't
1387                  * have RTF_MULTIRT set so we look for such an IRE in the
1388                  * bucket.
1389                  */
1390                 if (alt_ire->ire_flags & RTF_MULTIRT) {
1391                         irb_t           *irb;
1392                         ire_t           *ire1;

1394                         irb = ire->ire_bucket;
1395                         irb_refhold(irb);
1396                         for (ire1 = irb->irb_ire; ire1 != NULL;
1397                             ire1 = ire1->ire_next) {
1398                                 if (IRE_IS_CONDEMNED(ire1))
1399                                         continue;
1400                                 if (!(ire1->ire_type & IRE_BROADCAST) ||
1401                                     (ire1->ire_flags & RTF_MULTIRT))
1402                                         continue;
1403                                 ill = ire1->ire_ill;
1404                                 ill_refhold(ill);
1405                                 break;
1406                         }
1407                         irb_refrele(irb);
1408                         if (ire1 != NULL) {
1409                                 ill_t *orig_ill = ira->ira_ill;

1411                                 ire_refrele(alt_ire);
1412                                 /* Reassemble on the new ill */
1413                                 ira->ira_ill = ill;
1414                                 ip_input_local_v4(ire, mp, ipha, ira);
1415                                 ill_refrele(ill);
1416                                 /* Restore */
1417                                 ira->ira_ill = orig_ill;
1418                                 ira->ira_ruifindex =
1419                                     orig_ill->ill_phyint->phyint_ifindex;
1420                                 return;
1421                         }
1422                 }
1423                 ire_refrele(alt_ire);
1424                 /* Reassemble on the ill on which the packet arrived */
1425                 ip_input_local_v4(ire, mp, ipha, ira);
1426                 goto done;
1427         }

1429         /*
1430          * This is a directed broadcast
1431          *
1432          * If directed broadcast is allowed, then forward the packet out
1433          * the destination interface with IXAF_LOOPBACK_COPY set. That will
1434          * result in ip_input() receiving a copy of the packet on the
1435          * appropriate ill. (We could optimize this to avoid the extra trip
1436          * via ip_input(), but since directed broadcasts are normally disabled
1437          * it doesn't make sense to optimize it.)
1438          */
1439         if (!ipst->ips_ip_g_forward_directed_bcast ||
1440             (ira->ira_flags & (IRAF_L2DST_MULTICAST|IRAF_L2DST_BROADCAST))) {
1441                 ip_drop_input("directed broadcast not allowed", mp, ill);
1442                 freemsg(mp);
1443                 goto done;
1444         }
1445         if ((ira->ira_flags & IRAF_VERIFY_IP_CKSUM) && ip_csum_hdr(ipha)) {
1446                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1447                 ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
```

```
1448                freemsg(mp);
1449                goto done;
1450        }

1452        /*
1453         * Clear the indication that this may have hardware
1454         * checksum as we are not using it for forwarding.
1455         */
1456        DB_CKSUMFLAGS(mp) = 0;

1458        /*
1459         * Adjust ttl to 2 (1+1 - the forward engine will decrement it by one.
1460         */
1461        ipha->ipha_ttl = ipst->ips_ip_broadcast_ttl + 1;
1462        ipha->ipha_hdr_checksum = 0;
1463        ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

1465        /*
1466         * We use ip_forward_xmit to do any fragmentation.
1467         * and loopback copy on the outbound interface.
1468         *
1469         * Make it so that IXAF_LOOPBACK_COPY to be set on transmit side.
1470         */
1471        ira->ira_flags |= IRAF_LOOPBACK_COPY;

1473        nce = arp_nce_init(dst_ill, ipha->ipha_dst, IRE_BROADCAST);
1474        if (nce == NULL) {
1475                BUMP_MIB(dst_ill->ill_ip_mib, ipIfStatsOutDiscards);
1476                ip_drop_output("No nce", mp, dst_ill);
1477                freemsg(mp);
1478                goto done;
1479        }

1481        ip_forward_xmit_v4(nce, ill, mp, ipha, ira, dst_ill->ill_mc_mtu, 0);
1482        nce_refrele(nce);
1483 done:
1484        /* Restore */
1485        ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;
1486 }

1488 /*
1489  * ire_recvfn for IRE_MULTICAST.
1490  */
1491 void
1492 ire_recv_multicast_v4(ire_t *ire, mblk_t *mp, void *iph_arg,
1493     ip_recv_attr_t *ira)
1494 {
1495        ipha_t          *ipha = (ipha_t *)iph_arg;
1496        ill_t           *ill = ira->ira_ill;
1497        ip_stack_t      *ipst = ill->ill_ipst;

1499        ASSERT(ire->ire_ill == ira->ira_ill);

1501        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInMcastPkts);
1502        UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCInMcastOctets, ira->ira_pktlen);

1504        /* RSVP hook */
1505        if (ira->ira_flags & IRAF_RSVP)
1506                goto forus;

1508        /* Tag for higher-level protocols */
1509        ira->ira_flags |= IRAF_MULTICAST;

1511        /*
1512         * So that we don't end up with dups, only one ill an IPMP group is
1513         * nominated to receive multicast traffic.
```



```
1514         * If we have no cast_ill we are liberal and accept everything.
1515         */
1516        if (IS_UNDER_IPMP(ill)) {
1517                ip_stack_t      *ipst = ill->ill_ipst;

1519                /* For an under ill_grp can change under lock */
1520                rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1521                if (!ill->ill_nom_cast && ill->ill_grp != NULL &&
1522                    ill->ill_grp->ig_cast_ill != NULL) {
1523                        rw_exit(&ipst->ips_ill_g_lock);
1524                        ip_drop_input("not on cast ill", mp, ill);
1525                        freemsg(mp);
1526                        return;
1527                }
1528                rw_exit(&ipst->ips_ill_g_lock);
1529                /*
1530                 * We switch to the upper ill so that mrouter and hasmembers
1531                 * can operate on upper here and in ip_input_multicast.
1532                 */
1533                ill = ipmp_ill_hold_ipmp_ill(ill);
1534                if (ill != NULL) {
1535                        ASSERT(ill != ira->ira_ill);
1536                        ASSERT(ire->ire_ill == ira->ira_ill);
1537                        ira->ira_ill = ill;
1538                        ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;
1539                } else {
1540                        ill = ira->ira_ill;
1541                }
1542        }

1544        /*
1545         * Check if we are a multicast router - send ip_mforward a copy of
1546         * the packet.
1547         * Due to mroute_decap tunnels we consider forwarding packets even if
1548         * mrouted has not joined the allmulti group on this interface.
1549         */
1550        if (ipst->ips_ip_g_mrouter) {
1551                int retval;

1553                /*
1554                 * Clear the indication that this may have hardware
1555                 * checksum as we are not using it for forwarding.
1556                 */
1557                DB_CKSUMFLAGS(mp) = 0;

1559                /*
1560                 * ip_mforward helps us make these distinctions: If received
1561                 * on tunnel and not IGMP, then drop.
1562                 * If IGMP packet, then don't check membership
1563                 * If received on a phyint and IGMP or PIM, then
1564                 * don't check membership
1565                 */
1566                retval = ip_mforward(mp, ira);
1567                /* ip_mforward updates mib variables if needed */

1569                switch (retval) {
1570                case 0:
1571                        /*
1572                         * pkt is okay and arrived on phyint.
1573                         *
1574                         * If we are running as a multicast router
1575                         * we need to see all IGMP and/or PIM packets.
1576                         */
1577                        if ((ipha->ipha_protocol == IPPROTO_IGMP) ||
1578                            (ipha->ipha_protocol == IPPROTO_PIM)) {
1579                                goto forus;
```

```
1580                         }
1581                         break;
1582                 case -1:
1583                         /* pkt is mal-formed, toss it */
1584                         freemsg(mp);
1585                         goto done;
1586                 case 1:
1587                         /*
1588                          * pkt is okay and arrived on a tunnel
1589                          *
1590                          * If we are running a multicast router
1591                          * we need to see all igmp packets.
1592                          */
1593                         if (ipha->ipha_protocol == IPPROTO_IGMP) {
1594                                 goto forus;
1595                         }
1596                         ip_drop_input("Multicast on tunnel ignored", mp, ill);
1597                         freemsg(mp);
1598                         goto done;
1599                 }
1600         }

1602         /*
1603          * Check if we have members on this ill. This is not necessary for
1604          * correctness because even if the NIC/GLD had a leaky filter, we
1605          * filter before passing to each conn_t.
1606          */
1607         if (!ill_hasmembers_v4(ill, ipha->ipha_dst)) {
1608                 /*
1609                  * Nobody interested
1610                  *
1611                  * This might just be caused by the fact that
1612                  * multiple IP Multicast addresses map to the same
1613                  * link layer multicast - no need to increment counter!
1614                  */
1615                 ip_drop_input("Multicast with no members", mp, ill);
1616                 freemsg(mp);
1617                 goto done;
1618         }
1619 forus:
1620         ip2dbg(("ire_recv_multicast_v4: multicast for us: 0x%x\n",
1621             ntohl(ipha->ipha_dst)));

1623         /*
1624          * After reassembly and IPsec we will need to duplicate the
1625          * multicast packet for all matching zones on the ill.
1626          */
1627         ira->ira_zoneid = ALL_ZONES;

1629         /* Reassemble on the ill on which the packet arrived */
1630         ip_input_local_v4(ire, mp, ipha, ira);
1631 done:
1632         if (ill != ire->ire_ill) {
1633                 ill_refrele(ill);
1634                 ira->ira_ill = ire->ire_ill;
1635                 ira->ira_ruifindex = ira->ira_ill->ill_phyint->phyint_ifindex;
1636         }
1637 }

1639 /*
1640  * ire_recvfn for IRE_OFFLINK with RTF_MULTIRT.
1641  * Drop packets since we don't forward out multirt routes.
1642  */
1643 /* ARGSUSED */
1644 void
1645 ire_recv_multirt_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_recv_attr_t *ira)
```



```
1646 {
1647         ill_t           *ill = ira->ira_ill;

1649         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInNoRoutes);
1650         ip_drop_input("Not forwarding out MULTIRT", mp, ill);
1651         freemsg(mp);
1652 }

1654 /*
1655  * ire_recvfn for IRE_LOOPBACK. This is only used when a FW_HOOK
1656  * has rewritten the packet to have a loopback destination address (We
1657  * filter out packet with a loopback destination from arriving over the wire).
1658  * We don't know what zone to use, thus we always use the GLOBAL_ZONEID.
1659  */
1660 void
1661 ire_recv_loopback_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_recv_attr_t *ira)
1662 {
1663         ipha_t          *ipha = (ipha_t *)iph_arg;
1664         ill_t           *ill = ira->ira_ill;
1665         ill_t           *ire_ill = ire->ire_ill;

1667         ira->ira_zoneid = GLOBAL_ZONEID;

1669         /* Switch to the lo0 ill for further processing  */
1670         if (ire_ill != ill) {
1671                 /*
1672                  * Update ira_ill to be the ILL on which the IP address
1673                  * is hosted.
1674                  * No need to hold the ill since we have a hold on the ire
1675                  */
1676                 ASSERT(ira->ira_ill == ira->ira_rill);
1677                 ira->ira_ill = ire_ill;

1679                 ip_input_local_v4(ire, mp, ipha, ira);

1681                 /* Restore */
1682                 ASSERT(ira->ira_ill == ire_ill);
1683                 ira->ira_ill = ill;
1684                 return;

1686         }
1687         ip_input_local_v4(ire, mp, ipha, ira);
1688 }

1690 /*
1691  * ire_recvfn for IRE_LOCAL.
1692  */
1693 void
1694 ire_recv_local_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_recv_attr_t *ira)
1695 {
1696         ipha_t          *ipha = (ipha_t *)iph_arg;
1697         ill_t           *ill = ira->ira_ill;
1698         ill_t           *ire_ill = ire->ire_ill;

1700         /* Make a note for DAD that this address is in use */
1701         ire->ire_last_used_time = LBOLT_FASTPATH;

1703         /* Only target the IRE_LOCAL with the right zoneid. */
1704         ira->ira_zoneid = ire->ire_zoneid;

1706         /*
1707          * If the packet arrived on the wrong ill, we check that
1708          * this is ok.
1709          * If it is, then we ensure that we do the reassembly on
1710          * the ill on which the address is hosted. We keep ira_rill as
1711          * the one on which the packet arrived, so that IP_PKTINFO and
```

```
1712          * friends can report this.
1713          */
1714         if (ire_ill != ill) {
1715                 ire_t *new_ire;

1717                 new_ire = ip_check_multihome(&ipha->ipha_dst, ire, ill);
1718                 if (new_ire == NULL) {
1719                         /* Drop packet */
1720                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
1721                         ip_drop_input("ipIfStatsInForwProhibits", mp, ill);
1722                         freemsg(mp);
1723                         return;
1724                 }
1725                 /*
1726                  * Update ira_ill to be the ILL on which the IP address
1727                  * is hosted. No need to hold the ill since we have a
1728                  * hold on the ire. Note that we do the switch even if
1729                  * new_ire == ire (for IPMP, ire would be the one corresponding
1730                  * to the IPMP ill).
1731                  */
1732                 ASSERT(ira->ira_ill == ira->ira_rill);
1733                 ira->ira_ill = new_ire->ire_ill;

1735                 /* ira_ruifindex tracks the upper for ira_rill */
1736                 if (IS_UNDER_IPMP(ill))
1737                         ira->ira_ruifindex = ill_get_upper_ifindex(ill);

1739                 ip_input_local_v4(new_ire, mp, ipha, ira);

1741                 /* Restore */
1742                 ASSERT(ira->ira_ill == new_ire->ire_ill);
1743                 ira->ira_ill = ill;
1744                 ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;

1746                 if (new_ire != ire)
1747                         ire_refrele(new_ire);
1748                 return;
1749         }

1751         ip_input_local_v4(ire, mp, ipha, ira);
1752 }

1754 /*
1755  * Common function for packets arriving for the host. Handles
1756  * checksum verification, reassembly checks, etc.
1757  */
1758 static void
1759 ip_input_local_v4(ire_t *ire, mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
1760 {
1761         ill_t           *ill = ira->ira_ill;
1762         iaflags_t       iraflags = ira->ira_flags;

1764         /*
1765          * Verify IP header checksum. If the packet was AH or ESP then
1766          * this flag has already been cleared. Likewise if the packet
1767          * had a hardware checksum.
1768          */
1769         if ((iraflags & IRAF_VERIFY_IP_CKSUM) && ip_csum_hdr(ipha)) {
1770                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1771                 ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
1772                 freemsg(mp);
1773                 return;
1774         }

1776         if (iraflags & IRAF_IPV4_OPTIONS) {
1777                 if (!ip_input_local_options(mp, ipha, ira)) {
```



```
1778                         /* Error has been sent and mp consumed */
1779                         return;
1780                 }
1781                 /*
1782                  * Some old hardware does partial checksum by including the
1783                  * whole IP header, so the partial checksum value might have
1784                  * become invalid if any option in the packet have been
1785                  * updated. Always clear partial checksum flag here.
1786                  */
1787                 DB_CKSUMFLAGS(mp) &= ~HCK_PARTIALCKSUM;
1788         }

1790         /*
1791          * Is packet part of fragmented IP packet?
1792          * We compare against defined values in network byte order
1793          */
1794         if (ipha->ipha_fragment_offset_and_flags &
1795             (IPH_MF_HTONS | IPH_OFFSET_HTONS)) {
1796                 /*
1797                  * Make sure we have ira_l2src before we loose the original
1798                  * mblk
1799                  */
1800                 if (!(ira->ira_flags & IRAF_L2SRC_SET))
1801                         ip_setl2src(mp, ira, ira->ira_rill);

1803                 mp = ip_input_fragment(mp, ipha, ira);
1804                 if (mp == NULL)
1805                         return;
1806                 /* Completed reassembly */
1807                 ipha = (ipha_t *)mp->b_rptr;
1808         }

1810         /*
1811          * For broadcast and multicast we need some extra work before
1812          * we call ip_fanout_v4(), since in the case of shared-IP zones
1813          * we need to pretend that a packet arrived for each zoneid.
1814          */
1815         if (iraflags & IRAF_MULTIBROADCAST) {
1816                 if (iraflags & IRAF_BROADCAST)
1817                         ip_input_broadcast_v4(ire, mp, ipha, ira);
1818                 else
1819                         ip_input_multicast_v4(ire, mp, ipha, ira);
1820                 return;
1821         }
1822         ip_fanout_v4(mp, ipha, ira);
1823 }


1826 /*
1827  * Handle multiple zones which match the same broadcast address
1828  * and ill by delivering a packet to each of them.
1829  * Walk the bucket and look for different ire_zoneid but otherwise
1830  * the same IRE (same ill/addr/mask/type).
1831  * Note that ire_add() tracks IREs that are identical in all
1832  * fields (addr/mask/type/gw/ill/zoneid) within a single IRE by
1833  * increasing ire_identical_cnt. Thus we don't need to be concerned
1834  * about those.
1835  */
1836 static void
1837 ip_input_broadcast_v4(ire_t *ire, mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
1838 {
1839         ill_t           *ill = ira->ira_ill;
1840         ip_stack_t      *ipst = ill->ill_ipst;
1841         netstack_t      *ns = ipst->ips_netstack;
1842         irb_t           *irb;
1843         ire_t           *ire1;
```

```
1844         mblk_t          *mp1;
1845         ipha_t          *ipha1;
1846         uint_t          ira_pktlen = ira->ira_pktlen;
1847         uint16_t        ira_ip_hdr_length = ira->ira_ip_hdr_length;

1849         irb = ire->ire_bucket;

1851         /*
1852          * If we don't have more than one shared-IP zone, or if
1853          * there can't be more than one IRE_BROADCAST for this
1854          * IP address, then just set the zoneid and proceed.
1855          */
1856         if (ns->netstack_numzones == 1 || irb->irb_ire_cnt == 1) {
1857                 ira->ira_zoneid = ire->ire_zoneid;

1859                 ip_fanout_v4(mp, ipha, ira);
1860                 return;
1861         }
1862         irb_refhold(irb);
1863         for (ire1 = irb->irb_ire; ire1 != NULL; ire1 = ire1->ire_next) {
1864                 /* We do the main IRE after the end of the loop */
1865                 if (ire1 == ire)
1866                         continue;

1868                 /*
1869                  * Only IREs for the same IP address should be in the same
1870                  * bucket.
1871                  * But could have IRE_HOSTs in the case of CGTP.
1872                  */
1873                 ASSERT(ire1->ire_addr == ire->ire_addr);
1874                 if (!(ire1->ire_type & IRE_BROADCAST))
1875                         continue;

1877                 if (IRE_IS_CONDEMNED(ire1))
1878                         continue;

1880                 mp1 = copymsg(mp);
1881                 if (mp1 == NULL) {
1882                         /* Failed to deliver to one zone */
1883                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1884                         ip_drop_input("ipIfStatsInDiscards", mp, ill);
1885                         continue;
1886                 }
1887                 ira->ira_zoneid = ire1->ire_zoneid;
1888                 ipha1 = (ipha_t *)mp1->b_rptr;
1889                 ip_fanout_v4(mp1, ipha1, ira);
1890                 /*
1891                  * IPsec might have modified ira_pktlen and ira_ip_hdr_length
1892                  * so we restore them for a potential next iteration
1893                  */
1894                 ira->ira_pktlen = ira_pktlen;
1895                 ira->ira_ip_hdr_length = ira_ip_hdr_length;
1896         }
1897         irb_refrele(irb);
1898         /* Do the main ire */
1899         ira->ira_zoneid = ire->ire_zoneid;
1900         ip_fanout_v4(mp, ipha, ira);
1901 }

1903 /*
1904  * Handle multiple zones which want to receive the same multicast packets
1905  * on this ill by delivering a packet to each of them.
1906  *
1907  * Note that for packets delivered to transports we could instead do this
1908  * as part of the fanout code, but since we need to handle icmp_inbound
1909  * it is simpler to have multicast work the same as broadcast.
```



```
1910  *
1911  * The ip_fanout matching for multicast matches based on ilm independent of
1912  * zoneid since the zoneid restriction is applied when joining a multicast
1913  * group.
1914  */
1915 /* ARGSUSED */
1916 static void
1917 ip_input_multicast_v4(ire_t *ire, mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
1918 {
1919         ill_t           *ill = ira->ira_ill;
1920         iaflags_t       iraflags = ira->ira_flags;
1921         ip_stack_t      *ipst = ill->ill_ipst;
1922         netstack_t      *ns = ipst->ips_netstack;
1923         zoneid_t        zoneid;
1924         mblk_t          *mp1;
1925         ipha_t          *ipha1;
1926         uint_t          ira_pktlen = ira->ira_pktlen;
1927         uint16_t        ira_ip_hdr_length = ira->ira_ip_hdr_length;

1929         /* ire_recv_multicast has switched to the upper ill for IPMP */
1930         ASSERT(!IS_UNDER_IPMP(ill));

1932         /*
1933          * If we don't have more than one shared-IP zone, or if
1934          * there are no members in anything but the global zone,
1935          * then just set the zoneid and proceed.
1936          */
1937         if (ns->netstack_numzones == 1 ||
1938             !ill_hasmembers_otherzones_v4(ill, ipha->ipha_dst,
1939             GLOBAL_ZONEID)) {
1940                 ira->ira_zoneid = GLOBAL_ZONEID;

1942                 /* If sender didn't want this zone to receive it, drop */
1943                 if ((iraflags & IRAF_NO_LOOP_ZONEID_SET) &&
1944                     ira->ira_no_loop_zoneid == ira->ira_zoneid) {
1945                         ip_drop_input("Multicast but wrong zoneid", mp, ill);
1946                         freemsg(mp);
1947                         return;
1948                 }
1949                 ip_fanout_v4(mp, ipha, ira);
1950                 return;
1951         }

1953         /*
1954          * Here we loop over all zoneids that have members in the group
1955          * and deliver a packet to ip_fanout for each zoneid.
1956          *
1957          * First find any members in the lowest numeric zoneid by looking for
1958          * first zoneid larger than -1 (ALL_ZONES).
1959          * We terminate the loop when we receive -1 (ALL_ZONES).
1960          */
1961         zoneid = ill_hasmembers_nextzone_v4(ill, ipha->ipha_dst, ALL_ZONES);
1962         for (; zoneid != ALL_ZONES;
1963             zoneid = ill_hasmembers_nextzone_v4(ill, ipha->ipha_dst, zoneid)) {
1964                 /*
1965                  * Avoid an extra copymsg/freemsg by skipping global zone here
1966                  * and doing that at the end.
1967                  */
1968                 if (zoneid == GLOBAL_ZONEID)
1969                         continue;

1971                 ira->ira_zoneid = zoneid;

1973                 /* If sender didn't want this zone to receive it, skip */
1974                 if ((iraflags & IRAF_NO_LOOP_ZONEID_SET) &&
1975                     ira->ira_no_loop_zoneid == ira->ira_zoneid)
```

```
1976                        continue;

1978                mp1 = copymsg(mp);
1979                if (mp1 == NULL) {
1980                        /* Failed to deliver to one zone */
1981                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1982                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
1983                        continue;
1984                }
1985                ipha1 = (ipha_t *)mp1->b_rptr;
1986                ip_fanout_v4(mp1, ipha1, ira);
1987                /*
1988                 * IPsec might have modified ira_pktlen and ira_ip_hdr_length
1989                 * so we restore them for a potential next iteration
1990                 */
1991                ira->ira_pktlen = ira_pktlen;
1992                ira->ira_ip_hdr_length = ira_ip_hdr_length;
1993        }

1995        /* Do the main ire */
1996        ira->ira_zoneid = GLOBAL_ZONEID;
1997        /* If sender didn't want this zone to receive it, drop */
1998        if ((iraflags & IRAF_NO_LOOP_ZONEID_SET) &&
1999            ira->ira_no_loop_zoneid == ira->ira_zoneid) {
2000                ip_drop_input("Multicast but wrong zoneid", mp, ill);
2001                freemsg(mp);
2002        } else {
2003                ip_fanout_v4(mp, ipha, ira);
2004        }
2005 }


2008 /*
2009  * Determine the zoneid and IRAF_TX_* flags if trusted extensions
2010  * is in use. Updates ira_zoneid and ira_flags as a result.
2011  */
2012 static void
2013 ip_fanout_tx_v4(mblk_t *mp, ipha_t *ipha, uint8_t protocol,
2014     uint_t ip_hdr_length, ip_recv_attr_t *ira)
2015 {
2016        uint16_t        *up;
2017        uint16_t        lport;
2018        zoneid_t        zoneid;

2020        ASSERT(ira->ira_flags & IRAF_SYSTEM_LABELED);

2022        /*
2023         * If the packet is unlabeled we might allow read-down
2024         * for MAC_EXEMPT. Below we clear this if it is a multi-level
2025         * port (MLP).
2026         * Note that ira_tsl can be NULL here.
2027         */
2028        if (ira->ira_tsl != NULL && ira->ira_tsl->tsl_flags & TSLF_UNLABELED)
2029                ira->ira_flags |= IRAF_TX_MAC_EXEMPTABLE;

2031        if (ira->ira_zoneid != ALL_ZONES)
2032                return;

2034        ira->ira_flags |= IRAF_TX_SHARED_ADDR;

2036        up = (uint16_t *)((uchar_t *)ipha + ip_hdr_length);
2037        switch (protocol) {
2038        case IPPROTO_TCP:
2039        case IPPROTO_SCTP:
2040        case IPPROTO_UDP:
2041                /* Caller ensures this */
```



```
2042                ASSERT(((uchar_t *)ipha) + ip_hdr_length +4 <= mp->b_wptr);

2044                /*
2045                 * Only these transports support MLP.
2046                 * We know their destination port numbers is in
2047                 * the same place in the header.
2048                 */
2049                lport = up[1];

2051                /*
2052                 * No need to handle exclusive-stack zones
2053                 * since ALL_ZONES only applies to the shared IP instance.
2054                 */
2055                zoneid = tsol_mlp_findzone(protocol, lport);
2056                /*
2057                 * If no shared MLP is found, tsol_mlp_findzone returns
2058                 * ALL_ZONES.  In that case, we assume it's SLP, and
2059                 * search for the zone based on the packet label.
2060                 *
2061                 * If there is such a zone, we prefer to find a
2062                 * connection in it.  Otherwise, we look for a
2063                 * MAC-exempt connection in any zone whose label
2064                 * dominates the default label on the packet.
2065                 */
2066                if (zoneid == ALL_ZONES)
2067                        zoneid = tsol_attr_to_zoneid(ira);
2068                else
2069                        ira->ira_flags &= ~IRAF_TX_MAC_EXEMPTABLE;
2070                break;
2071        default:
2072                /* Handle shared address for other protocols */
2073                zoneid = tsol_attr_to_zoneid(ira);
2074                break;
2075        }
2076        ira->ira_zoneid = zoneid;
2077 }

2079 /*
2080  * Increment checksum failure statistics
2081  */
2082 static void
2083 ip_input_cksum_err_v4(uint8_t protocol, uint16_t hck_flags, ill_t *ill)
2084 {
2085        ip_stack_t      *ipst = ill->ill_ipst;

2087        switch (protocol) {
2088        case IPPROTO_TCP:
2089                BUMP_MIB(ill->ill_ip_mib, tcpIfStatsInErrs);

2091                if (hck_flags & HCK_FULLCKSUM)
2092                        IP_STAT(ipst, ip_tcp_in_full_hw_cksum_err);
2093                else if (hck_flags & HCK_PARTIALCKSUM)
2094                        IP_STAT(ipst, ip_tcp_in_part_hw_cksum_err);
2095                else
2096                        IP_STAT(ipst, ip_tcp_in_sw_cksum_err);
2097                break;
2098        case IPPROTO_UDP:
2099                BUMP_MIB(ill->ill_ip_mib, udpIfStatsInCksumErrs);
2100                if (hck_flags & HCK_FULLCKSUM)
2101                        IP_STAT(ipst, ip_udp_in_full_hw_cksum_err);
2102                else if (hck_flags & HCK_PARTIALCKSUM)
2103                        IP_STAT(ipst, ip_udp_in_part_hw_cksum_err);
2104                else
2105                        IP_STAT(ipst, ip_udp_in_sw_cksum_err);
2106                break;
2107        case IPPROTO_ICMP:
```

```
2108                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInCksumErrs);
2109                 break;
2110         default:
2111                 ASSERT(0);
2112                 break;
2113         }
2114 }

2116 /* Calculate the IPv4 pseudo-header checksum */
2117 uint32_t
2118 ip_input_cksum_pseudo_v4(ipha_t *ipha, ip_recv_attr_t *ira)
2119 {
2120         uint_t          ulp_len;
2121         uint32_t        cksum;
2122         uint8_t         protocol = ira->ira_protocol;
2123         uint16_t        ip_hdr_length = ira->ira_ip_hdr_length;

2125 #define iphs    ((uint16_t *)ipha)

2127         switch (protocol) {
2128         case IPPROTO_TCP:
2129                 ulp_len = ira->ira_pktlen - ip_hdr_length;

2131                 /* Protocol and length */
2132                 cksum = htons(ulp_len) + IP_TCP_CSUM_COMP;
2133                 /* IP addresses */
2134                 cksum += iphs[6] + iphs[7] + iphs[8] + iphs[9];
2135                 break;

2137         case IPPROTO_UDP: {
2138                 udpha_t         *udpha;

2140                 udpha = (udpha_t  *)((uchar_t *)ipha + ip_hdr_length);

2142                 /* Protocol and length */
2143                 cksum = udpha->uha_length + IP_UDP_CSUM_COMP;
2144                 /* IP addresses */
2145                 cksum += iphs[6] + iphs[7] + iphs[8] + iphs[9];
2146                 break;
2147         }

2149         default:
2150                 cksum = 0;
2151                 break;
2152         }
2153 #undef  iphs
2154         return (cksum);
2155 }


2158 /*
2159  * Software verification of the ULP checksums.
2160  * Returns B_TRUE if ok.
2161  * Increments statistics of failed.
2162  */
2163 static boolean_t
2164 ip_input_sw_cksum_v4(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
2165 {
2166         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
2167         uint32_t        cksum;
2168         uint8_t         protocol = ira->ira_protocol;
2169         uint16_t        ip_hdr_length = ira->ira_ip_hdr_length;

2171         IP_STAT(ipst, ip_in_sw_cksum);

2173         ASSERT(protocol == IPPROTO_TCP || protocol == IPPROTO_UDP);
```



```
2175         cksum = ip_input_cksum_pseudo_v4(ipha, ira);
2176         cksum = IP_CSUM(mp, ip_hdr_length, cksum);
2177         if (cksum == 0)
2178                 return (B_TRUE);

2180         ip_input_cksum_err_v4(protocol, 0, ira->ira_ill);
2181         return (B_FALSE);
2182 }

2184 /*
2185  * Verify the ULP checksums.
2186  * Returns B_TRUE if ok, or if the ULP doesn't have a well-defined checksum
2187  * algorithm.
2188  * Increments statistics if failed.
2189  */
2190 static boolean_t
2191 ip_input_cksum_v4(iaflags_t iraflags, mblk_t *mp, ipha_t *ipha,
2192     ip_recv_attr_t *ira)
2193 {
2194         ill_t           *ill = ira->ira_rill;
2195         uint16_t        hck_flags;
2196         uint32_t        cksum;
2197         mblk_t          *mp1;
2198         int32_t         len;
2199         uint8_t         protocol = ira->ira_protocol;
2200         uint16_t        ip_hdr_length = ira->ira_ip_hdr_length;


2203         switch (protocol) {
2204         case IPPROTO_TCP:
2205                 break;

2207         case IPPROTO_UDP: {
2208                 udpha_t         *udpha;

2210                 udpha = (udpha_t  *)((uchar_t *)ipha + ip_hdr_length);
2211                 if (udpha->uha_checksum == 0) {
2212                         /* Packet doesn't have a UDP checksum */
2213                         return (B_TRUE);
2214                 }
2215                 break;
2216         }
2217         case IPPROTO_SCTP: {
2218                 sctp_hdr_t      *sctph;
2219                 uint32_t        pktsum;

2221                 sctph = (sctp_hdr_t *)((uchar_t *)ipha + ip_hdr_length);
2222 #ifdef  DEBUG
2223                 if (skip_sctp_cksum)
2224                         return (B_TRUE);
2225 #endif
2226                 pktsum = sctph->sh_chksum;
2227                 sctph->sh_chksum = 0;
2228                 cksum = sctp_cksum(mp, ip_hdr_length);
2229                 sctph->sh_chksum = pktsum;
2230                 if (cksum == pktsum)
2231                         return (B_TRUE);

2233                 /*
2234                  * Defer until later whether a bad checksum is ok
2235                  * in order to allow RAW sockets to use Adler checksum
2236                  * with SCTP.
2237                  */
2238                 ira->ira_flags |= IRAF_SCTP_CSUM_ERR;
2239                 return (B_TRUE);
```

```
2240         }

2242         default:
2243                 /* No ULP checksum to verify. */
2244                 return (B_TRUE);
2245         }
2246         /*
2247          * Revert to software checksum calculation if the interface
2248          * isn't capable of checksum offload.
2249          * We clear DB_CKSUMFLAGS when going through IPsec in ip_fanout.
2250          * Note: IRAF_NO_HW_CKSUM is not currently used.
2251          */
2252         ASSERT(!IS_IPMP(ill));
2253         if ((iraflags & IRAF_NO_HW_CKSUM) || !ILL_HCKSUM_CAPABLE(ill) ||
2254             !dohwcksum) {
2255                 return (ip_input_sw_cksum_v4(mp, ipha, ira));
2256         }

2258         /*
2259          * We apply this for all ULP protocols. Does the HW know to
2260          * not set the flags for SCTP and other protocols.
2261          */

2263         hck_flags = DB_CKSUMFLAGS(mp);

2265         if (hck_flags & HCK_FULLCKSUM_OK) {
2266                 /*
2267                  * Hardware has already verified the checksum.
2268                  */
2269                 return (B_TRUE);
2270         }

2272         if (hck_flags & HCK_FULLCKSUM) {
2273                 /*
2274                  * Full checksum has been computed by the hardware
2275                  * and has been attached.  If the driver wants us to
2276                  * verify the correctness of the attached value, in
2277                  * order to protect against faulty hardware, compare
2278                  * it against -0 (0xFFFF) to see if it's valid.
2279                  */
2280                 cksum = DB_CKSUM16(mp);
2281                 if (cksum == 0xFFFF)
2282                         return (B_TRUE);
2283                 ip_input_cksum_err_v4(protocol, hck_flags, ira->ira_ill);
2284                 return (B_FALSE);
2285         }

2287         mp1 = mp->b_cont;
2288         if ((hck_flags & HCK_PARTIALCKSUM) &&
2289             (mp1 == NULL || mp1->b_cont == NULL) &&
2290             ip_hdr_length >= DB_CKSUMSTART(mp) &&
2291             ((len = ip_hdr_length - DB_CKSUMSTART(mp)) & 1) == 0) {
2292                 uint32_t        adj;
2293                 uchar_t         *cksum_start;

2295                 cksum = ip_input_cksum_pseudo_v4(ipha, ira);

2297                 cksum_start = ((uchar_t *)ipha + DB_CKSUMSTART(mp));

2299                 /*
2300                  * Partial checksum has been calculated by hardware
2301                  * and attached to the packet; in addition, any
2302                  * prepended extraneous data is even byte aligned,
2303                  * and there are at most two mblks associated with
2304                  * the packet.  If any such data exists, we adjust
2305                  * the checksum; also take care any postpended data.
```



```
2306                  */
2307                 IP_ADJCKSUM_PARTIAL(cksum_start, mp, mp1, len, adj);
2308                 /*
2309                  * One's complement subtract extraneous checksum
2310                  */
2311                 cksum += DB_CKSUM16(mp);
2312                 if (adj >= cksum)
2313                         cksum = ~(adj - cksum) & 0xFFFF;
2314                 else
2315                         cksum -= adj;
2316                 cksum = (cksum & 0xFFFF) + ((int)cksum >> 16);
2317                 cksum = (cksum & 0xFFFF) + ((int)cksum >> 16);
2318                 if (!(~cksum & 0xFFFF))
2319                         return (B_TRUE);

2321                 ip_input_cksum_err_v4(protocol, hck_flags, ira->ira_ill);
2322                 return (B_FALSE);
2323         }
2324         return (ip_input_sw_cksum_v4(mp, ipha, ira));
2325 }


2328 /*
2329  * Handle fanout of received packets.
2330  * Unicast packets that are looped back (from ire_send_local_v4) and packets
2331  * from the wire are differentiated by checking IRAF_VERIFY_ULP_CKSUM.
2332  *
2333  * IPQoS Notes
2334  * Before sending it to the client, invoke IPPF processing. Policy processing
2335  * takes place only if the callout_position, IPP_LOCAL_IN, is enabled.
2336  */
2337 void
2338 ip_fanout_v4(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
2339 {
2340         ill_t           *ill = ira->ira_ill;
2341         iaflags_t       iraflags = ira->ira_flags;
2342         ip_stack_t      *ipst = ill->ill_ipst;
2343         uint8_t         protocol = ipha->ipha_protocol;
2344         conn_t          *connp;
2345 #define rptr    ((uchar_t *)ipha)
2346         uint_t          ip_hdr_length;
2347         uint_t          min_ulp_header_length;
2348         int             offset;
2349         ssize_t         len;
2350         netstack_t      *ns = ipst->ips_netstack;
2351         ipsec_stack_t   *ipss = ns->netstack_ipsec;
2352         ill_t           *rill = ira->ira_rill;

2354         ASSERT(ira->ira_pktlen == ntohs(ipha->ipha_length));

2356         ip_hdr_length = ira->ira_ip_hdr_length;
2357         ira->ira_protocol = protocol;

2359         /*
2360          * Time for IPP once we've done reassembly and IPsec.
2361          * We skip this for loopback packets since we don't do IPQoS
2362          * on loopback.
2363          */
2364         if (IPP_ENABLED(IPP_LOCAL_IN, ipst) &&
2365             !(iraflags & IRAF_LOOPBACK) &&
2366             (protocol != IPPROTO_ESP || protocol != IPPROTO_AH)) {
2367                 /*
2368                  * Use the interface on which the packet arrived - not where
2369                  * the IP address is hosted.
2370                  */
2371                 /* ip_process translates an IS_UNDER_IPMP */
```

```
2372                mp = ip_process(IPP_LOCAL_IN, mp, rill, ill);
2373                if (mp == NULL) {
2374                        /* ip_drop_packet and MIB done */
2375                        return;
2376                }
2377        }

2379        /* Determine the minimum required size of the upper-layer header */
2380        /* Need to do this for at least the set of ULPs that TX handles. */
2381        switch (protocol) {
2382        case IPPROTO_TCP:
2383                min_ulp_header_length = TCP_MIN_HEADER_LENGTH;
2384                break;
2385        case IPPROTO_SCTP:
2386                min_ulp_header_length = SCTP_COMMON_HDR_LENGTH;
2387                break;
2388        case IPPROTO_UDP:
2389                min_ulp_header_length = UDPH_SIZE;
2390                break;
2391        case IPPROTO_ICMP:
2392                min_ulp_header_length = ICMPH_SIZE;
2393                break;
2394        case IPPROTO_DCCP:
2395                min_ulp_header_length = DCCP_MIN_HEADER_LENGTH;
2396                break;
2397 #endif /* ! codereview */
2398        default:
2399                min_ulp_header_length = 0;
2400                break;
2401        }
2402        /* Make sure we have the min ULP header length */
2403        len = mp->b_wptr - rptr;
2404        if (len < ip_hdr_length + min_ulp_header_length) {
2405                if (ira->ira_pktlen < ip_hdr_length + min_ulp_header_length) {
2406                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
2407                        ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
2408                        freemsg(mp);
2409                        return;
2410                }
2411                IP_STAT(ipst, ip_recv_pullup);
2412                ipha = ip_pullup(mp, ip_hdr_length + min_ulp_header_length,
2413                    ira);
2414                if (ipha == NULL)
2415                        goto discard;
2416                len = mp->b_wptr - rptr;
2417        }

2419        /*
2420         * If trusted extensions then determine the zoneid and TX specific
2421         * ira_flags.
2422         */
2423        if (iraflags & IRAF_SYSTEM_LABELED) {
2424                /* This can update ira->ira_flags and ira->ira_zoneid */
2425                ip_fanout_tx_v4(mp, ipha, protocol, ip_hdr_length, ira);
2426                iraflags = ira->ira_flags;
2427        }


2430        /* Verify ULP checksum. Handles TCP, UDP, and SCTP */
2431        if (iraflags & IRAF_VERIFY_ULP_CKSUM) {
2432                if (!ip_input_cksum_v4(iraflags, mp, ipha, ira)) {
2433                        /* Bad checksum. Stats are already incremented */
2434                        ip_drop_input("Bad ULP checksum", mp, ill);
2435                        freemsg(mp);
2436                        return;
2437                }
```



```
2438                /* IRAF_SCTP_CSUM_ERR could have been set */
2439                iraflags = ira->ira_flags;
2440        }
2441        switch (protocol) {
2442        case IPPROTO_TCP:
2443                /* For TCP, discard broadcast and multicast packets. */
2444                if (iraflags & IRAF_MULTIBROADCAST)
2445                        goto discard;

2447                /* First mblk contains IP+TCP headers per above check */
2448                ASSERT(len >= ip_hdr_length + TCP_MIN_HEADER_LENGTH);

2450                /* TCP options present? */
2451                offset = ((uchar_t *)ipha)[ip_hdr_length + 12] >> 4;
2452                if (offset != 5) {
2453                        if (offset < 5)
2454                                goto discard;

2456                        /*
2457                         * There must be TCP options.
2458                         * Make sure we can grab them.
2459                         */
2460                        offset <<= 2;
2461                        offset += ip_hdr_length;
2462                        if (len < offset) {
2463                                if (ira->ira_pktlen < offset) {
2464                                        BUMP_MIB(ill->ill_ip_mib,
2465                                            ipIfStatsInTruncatedPkts);
2466                                        ip_drop_input(
2467                                            "ipIfStatsInTruncatedPkts",
2468                                            mp, ill);
2469                                        freemsg(mp);
2470                                        return;
2471                                }
2472                                IP_STAT(ipst, ip_recv_pullup);
2473                                ipha = ip_pullup(mp, offset, ira);
2474                                if (ipha == NULL)
2475                                        goto discard;
2476                                len = mp->b_wptr - rptr;
2477                        }
2478                }

2480                /*
2481                 * Pass up a squeue hint to tcp.
2482                 * If ira_sqp is already set (this is loopback) we leave it
2483                 * alone.
2484                 */
2485                if (ira->ira_sqp == NULL) {
2486                        ira->ira_sqp = ip_squeue_get(ira->ira_ring);
2487                }

2489                /* Look for AF_INET or AF_INET6 that matches */
2490                connp = ipcl_classify_v4(mp, IPPROTO_TCP, ip_hdr_length,
2491                    ira, ipst);
2492                if (connp == NULL) {
2493                        /* Send the TH_RST */
2494                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2495                        tcp_xmit_listeners_reset(mp, ira, ipst, NULL);
2496                        return;
2497                }
2498                if (connp->conn_incoming_ifindex != 0 &&
2499                    connp->conn_incoming_ifindex != ira->ira_ruifindex) {
2500                        CONN_DEC_REF(connp);

2502                        /* Send the TH_RST */
2503                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
```

```
2504                    tcp_xmit_listeners_reset(mp, ira, ipst, NULL);
2505                    return;
2506            }
2507            if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2508                (iraflags & IRAF_IPSEC_SECURE)) {
2509                mp = ipsec_check_inbound_policy(mp, connp,
2510                    ipha, NULL, ira);
2511                if (mp == NULL) {
2512                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2513                        /* Note that mp is NULL */
2514                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
2515                        CONN_DEC_REF(connp);
2516                        return;
2517                }
2518            }
2519            /* Found a client; up it goes */
2520            BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2521            ira->ira_ill = ira->ira_rill = NULL;
2522            if (!IPCL_IS_TCP(connp)) {
2523                    /* Not TCP; must be SOCK_RAW, IPPROTO_TCP */
2524                    (connp->conn_recv)(connp, mp, NULL, ira);
2525                    CONN_DEC_REF(connp);
2526                    ira->ira_ill = ill;
2527                    ira->ira_rill = rill;
2528                    return;
2529            }

2531            /*
2532             * We do different processing whether called from
2533             * ip_accept_tcp and we match the target, don't match
2534             * the target, and when we are called by ip_input.
2535             */
2536            if (iraflags & IRAF_TARGET_SQP) {
2537                    if (ira->ira_target_sqp == connp->conn_sqp) {
2538                            mblk_t  *attrmp;

2540                            attrmp = ip_recv_attr_to_mblk(ira);
2541                            if (attrmp == NULL) {
2542                                    BUMP_MIB(ill->ill_ip_mib,
2543                                        ipIfStatsInDiscards);
2544                                    ip_drop_input("ipIfStatsInDiscards",
2545                                        mp, ill);
2546                                    freemsg(mp);
2547                                    CONN_DEC_REF(connp);
2548                            } else {
2549                                    SET_SQUEUE(attrmp, connp->conn_recv,
2550                                        connp);
2551                                    attrmp->b_cont = mp;
2552                                    ASSERT(ira->ira_target_sqp_mp == NULL);
2553                                    ira->ira_target_sqp_mp = attrmp;
2554                                    /*
2555                                     * Conn ref release when drained from
2556                                     * the squeue.
2557                                     */
2558                            }
2559                    } else {
2560                            SQUEUE_ENTER_ONE(connp->conn_sqp, mp,
2561                                connp->conn_recv, connp, ira, SQ_FILL,
2562                                SQTAG_IP_TCP_INPUT);
2563                    }
2564            } else {
2565                    SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_recv,
2566                        connp, ira, ip_squeue_flag, SQTAG_IP_TCP_INPUT);
2567            }
2568            ira->ira_ill = ill;
2569            ira->ira_rill = rill;
```



```
2570                    return;

2572            case IPPROTO_SCTP: {
2573                    sctp_hdr_t      *sctph;
2574                    in6_addr_t      map_src, map_dst;
2575                    uint32_t        ports;  /* Source and destination ports */
2576                    sctp_stack_t    *sctps = ipst->ips_netstack->netstack_sctp;

2578                    /* For SCTP, discard broadcast and multicast packets. */
2579                    if (iraflags & IRAF_MULTIBROADCAST)
2580                            goto discard;

2582                    /*
2583                     * Since there is no SCTP h/w cksum support yet, just
2584                     * clear the flag.
2585                     */
2586                    DB_CKSUMFLAGS(mp) = 0;

2588                    /* Length ensured above */
2589                    ASSERT(MBLKL(mp) >= ip_hdr_length + SCTP_COMMON_HDR_LENGTH);
2590                    sctph = (sctp_hdr_t *)(rptr + ip_hdr_length);

2592                    /* get the ports */
2593                    ports = *(uint32_t *)&sctph->sh_sport;

2595                    IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &map_dst);
2596                    IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &map_src);
2597                    if (iraflags & IRAF_SCTP_CSUM_ERR) {
2598                            /*
2599                             * No potential sctp checksum errors go to the Sun
2600                             * sctp stack however they might be Adler-32 summed
2601                             * packets a userland stack bound to a raw IP socket
2602                             * could reasonably use. Note though that Adler-32 is
2603                             * a long deprecated algorithm and customer sctp
2604                             * networks should eventually migrate to CRC-32 at
2605                             * which time this facility should be removed.
2606                             */
2607                            ip_fanout_sctp_raw(mp, ipha, NULL, ports, ira);
2608                            return;
2609                    }
2610                    connp = sctp_fanout(&map_src, &map_dst, ports, ira, mp,
2611                        sctps, sctph);
2612                    if (connp == NULL) {
2613                            /* Check for raw socket or OOTB handling */
2614                            ip_fanout_sctp_raw(mp, ipha, NULL, ports, ira);
2615                            return;
2616                    }
2617                    if (connp->conn_incoming_ifindex != 0 &&
2618                        connp->conn_incoming_ifindex != ira->ira_ruifindex) {
2619                            CONN_DEC_REF(connp);
2620                            /* Check for raw socket or OOTB handling */
2621                            ip_fanout_sctp_raw(mp, ipha, NULL, ports, ira);
2622                            return;
2623                    }

2625                    /* Found a client; up it goes */
2626                    BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2627                    sctp_input(connp, ipha, NULL, mp, ira);
2628                    /* sctp_input does a rele of the sctp_t */
2629                    return;
2630            }

2632            case IPPROTO_UDP:
2633                    /* First mblk contains IP+UDP headers as checked above */
2634                    ASSERT(MBLKL(mp) >= ip_hdr_length + UDPH_SIZE);
```

```
2636                if (iraflags & IRAF_MULTIBROADCAST) {
2637                        uint16_t *up;   /* Pointer to ports in ULP header */

2639                        up = (uint16_t *)((uchar_t *)ipha + ip_hdr_length);
2640                        ip_fanout_udp_multi_v4(mp, ipha, up[1], up[0], ira);
2641                        return;
2642                }

2644                /* Look for AF_INET or AF_INET6 that matches */
2645                connp = ipcl_classify_v4(mp, IPPROTO_UDP, ip_hdr_length,
2646                    ira, ipst);
2647                if (connp == NULL) {
2648        no_udp_match:
2649                        if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_UDP].
2650                            connf_head != NULL) {
2651                                ASSERT(ira->ira_protocol == IPPROTO_UDP);
2652                                ip_fanout_proto_v4(mp, ipha, ira);
2653                        } else {
2654                                ip_fanout_send_icmp_v4(mp,
2655                                    ICMP_DEST_UNREACHABLE,
2656                                    ICMP_PORT_UNREACHABLE, ira);
2657                        }
2658                        return;

2660                }
2661                if (connp->conn_incoming_ifindex != 0 &&
2662                    connp->conn_incoming_ifindex != ira->ira_ruifindex) {
2663                        CONN_DEC_REF(connp);
2664                        goto no_udp_match;
2665                }
2666                if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrld :
2667                    !canputnext(connp->conn_rq)) {
2668                        CONN_DEC_REF(connp);
2669                        BUMP_MIB(ill->ill_ip_mib, udpIfStatsInOverflows);
2670                        ip_drop_input("udpIfStatsInOverflows", mp, ill);
2671                        freemsg(mp);
2672                        return;
2673                }
2674                if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2675                    (iraflags & IRAF_IPSEC_SECURE)) {
2676                        mp = ipsec_check_inbound_policy(mp, connp,
2677                            ipha, NULL, ira);
2678                        if (mp == NULL) {
2679                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2680                                /* Note that mp is NULL */
2681                                ip_drop_input("ipIfStatsInDiscards", mp, ill);
2682                                CONN_DEC_REF(connp);
2683                                return;
2684                        }
2685                }
2686                /*
2687                 * Remove 0-spi if it's 0, or move everything behind
2688                 * the UDP header over it and forward to ESP via
2689                 * ip_fanout_v4().
2690                 */
2691                if (connp->conn_udp->udp_nat_t_endpoint) {
2692                        if (iraflags & IRAF_IPSEC_SECURE) {
2693                                ip_drop_packet(mp, B_TRUE, ira->ira_ill,
2694                                    DROPPER(ipss, ipds_esp_nat_t_ipsec),
2695                                    &ipss->ipsec_dropper);
2696                                CONN_DEC_REF(connp);
2697                                return;
2698                        }

2700                        mp = zero_spi_check(mp, ira);
2701                        if (mp == NULL) {
```



```
2702                                /*
2703                                 * Packet was consumed - probably sent to
2704                                 * ip_fanout_v4.
2705                                 */
2706                                CONN_DEC_REF(connp);
2707                                return;
2708                        }
2709                        /* Else continue like a normal UDP packet. */
2710                        ipha = (ipha_t *)mp->b_rptr;
2711                        protocol = ipha->ipha_protocol;
2712                        ira->ira_protocol = protocol;
2713                }
2714                /* Found a client; up it goes */
2715                IP_STAT(ipst, ip_udp_fannorm);
2716                BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2717                ira->ira_ill = ira->ira_rill = NULL;
2718                (connp->conn_recv)(connp, mp, NULL, ira);
2719                CONN_DEC_REF(connp);
2720                ira->ira_ill = ill;
2721                ira->ira_rill = rill;
2722                return;
2723        case IPPROTO_DCCP:
2724                /* For DCCP, discard broadcast and multicast packets */
2725                if (iraflags & IRAF_MULTIBROADCAST) {
2726                        goto discard;
2727                }

2729                /* First mblk contains IP+DCCP headers per above check */
2730                ASSERT(len >= ip_hdr_length + DCCP_MIN_HEADER_LENGTH);

2732                /* XXX:DCCP valid options */

2734                /* Squeue hint */
2735                if (ira->ira_sqp == NULL) {
2736                        ira->ira_sqp = ip_squeue_get(ira->ira_ring);
2737                }

2739                connp = ipcl_classify_v4(mp, IPPROTO_DCCP, ip_hdr_length,
2740                    ira, ipst);
2741                if (connp == NULL) {
2742                        cmn_err(CE_NOTE, "ip_input.c: ip_fanout_v4 connp not fou
2743                        /* Send the reset packet */
2744                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2745                        dccp_xmit_listeners_reset(mp, ira, ipst, NULL);
2746                        return;
2747                }

2749                if (connp->conn_incoming_ifindex != 0 &&
2750                    connp->conn_incoming_ifindex != ira->ira_ruifindex) {
2751                        cmn_err(CE_NOTE, "ip_input.c: ip_fanout_v4 ifindex probl
2752                        /* Send the reset packet */
2753                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2754                        dccp_xmit_listeners_reset(mp, ira, ipst, NULL);
2755                        return;
2756                }

2758                if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2759                    (iraflags & IRAF_IPSEC_SECURE)) {
2760                        mp = ipsec_check_inbound_policy(mp, connp,
2761                            ipha, NULL, ira);
2762                        if (mp == NULL) {
2763                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2764                                /* Note that mp is NULL */
2765                                ip_drop_input("ipIfStatsInDiscards", mp, ill);
2766                                CONN_DEC_REF(connp);
2767                                return;
```

```
2768                        }
2769                }

2771                /* Found a client; up it goes */
2772                BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2773                ira->ira_ill = ira->ira_rill = NULL;

2775                /* XXX SOCK_RAW for DCCP? */

2777                SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_recv,
2778                    connp, ira, ip_squeue_flag, SQTAG_IP_DCCP_INPUT);

2780                ira->ira_ill = ill;
2781                ira->ira_rill = rill;
2782                return;
2783 #endif /* ! codereview */
2784        default:
2785                break;
2786        }

2788        /*
2789         * Clear hardware checksumming flag as it is currently only
2790         * used by TCP and UDP.
2791         */
2792        DB_CKSUMFLAGS(mp) = 0;

2794        switch (protocol) {
2795        case IPPROTO_ICMP:
2796                /*
2797                 * We need to accomodate icmp messages coming in clear
2798                 * until we get everything secure from the wire. If
2799                 * icmp_accept_clear_messages is zero we check with
2800                 * the global policy and act accordingly. If it is
2801                 * non-zero, we accept the message without any checks.
2802                 * But *this does not mean* that this will be delivered
2803                 * to RAW socket clients. By accepting we might send
2804                 * replies back, change our MTU value etc.,
2805                 * but delivery to the ULP/clients depends on their
2806                 * policy dispositions.
2807                 */
2808                if (ipst->ips_icmp_accept_clear_messages == 0) {
2809                        mp = ipsec_check_global_policy(mp, NULL,
2810                            ipha, NULL, ira, ns);
2811                        if (mp == NULL)
2812                                return;
2813                }

2815                /*
2816                 * On a labeled system, we have to check whether the zone
2817                 * itself is permitted to receive raw traffic.
2818                 */
2819                if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
2820                        if (!tsol_can_accept_raw(mp, ira, B_FALSE)) {
2821                                BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
2822                                ip_drop_input("tsol_can_accept_raw", mp, ill);
2823                                freemsg(mp);
2824                                return;
2825                        }
2826                }

2828                /*
2829                 * ICMP header checksum, including checksum field,
2830                 * should be zero.
2831                 */
2832                if (IP_CSUM(mp, ip_hdr_length, 0)) {
2833                        BUMP_MIB(&ipst->ips_icmp_mib, icmpInCksumErrs);
```



```
2834                        ip_drop_input("icmpInCksumErrs", mp, ill);
2835                        freemsg(mp);
2836                        return;
2837                }
2838                BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2839                mp = icmp_inbound_v4(mp, ira);
2840                if (mp == NULL) {
2841                        /* No need to pass to RAW sockets */
2842                        return;
2843                }
2844                break;

2846        case IPPROTO_IGMP:
2847                /*
2848                 * If we are not willing to accept IGMP packets in clear,
2849                 * then check with global policy.
2850                 */
2851                if (ipst->ips_igmp_accept_clear_messages == 0) {
2852                        mp = ipsec_check_global_policy(mp, NULL,
2853                            ipha, NULL, ira, ns);
2854                        if (mp == NULL)
2855                                return;
2856                }
2857                if ((ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2858                    !tsol_can_accept_raw(mp, ira, B_TRUE)) {
2859                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2860                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
2861                        freemsg(mp);
2862                        return;
2863                }
2864                /*
2865                 * Validate checksum
2866                 */
2867                if (IP_CSUM(mp, ip_hdr_length, 0)) {
2868                        ++ipst->ips_igmpstat.igps_rcv_badsum;
2869                        ip_drop_input("igps_rcv_badsum", mp, ill);
2870                        freemsg(mp);
2871                        return;
2872                }

2874                BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2875                mp = igmp_input(mp, ira);
2876                if (mp == NULL) {
2877                        /* Bad packet - discarded by igmp_input */
2878                        return;
2879                }
2880                break;
2881        case IPPROTO_PIM:
2882                /*
2883                 * If we are not willing to accept PIM packets in clear,
2884                 * then check with global policy.
2885                 */
2886                if (ipst->ips_pim_accept_clear_messages == 0) {
2887                        mp = ipsec_check_global_policy(mp, NULL,
2888                            ipha, NULL, ira, ns);
2889                        if (mp == NULL)
2890                                return;
2891                }
2892                if ((ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2893                    !tsol_can_accept_raw(mp, ira, B_TRUE)) {
2894                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2895                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
2896                        freemsg(mp);
2897                        return;
2898                }
2899                BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
```

```
2901                /* Checksum is verified in pim_input */
2902                mp = pim_input(mp, ira);
2903                if (mp == NULL) {
2904                        /* Bad packet - discarded by pim_input */
2905                        return;
2906                }
2907                break;
2908        case IPPROTO_AH:
2909        case IPPROTO_ESP: {
2910                /*
2911                 * Fast path for AH/ESP.
2912                 */
2913                netstack_t *ns = ipst->ips_netstack;
2914                ipsec_stack_t *ipss = ns->netstack_ipsec;

2916                IP_STAT(ipst, ipsec_proto_ahesp);

2918                if (!ipsec_loaded(ipss)) {
2919                        ip_proto_not_sup(mp, ira);
2920                        return;
2921                }

2923                BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2924                /* select inbound SA and have IPsec process the pkt */
2925                if (protocol == IPPROTO_ESP) {
2926                        esph_t *esph;
2927                        boolean_t esp_in_udp_sa;
2928                        boolean_t esp_in_udp_packet;

2930                        mp = ipsec_inbound_esp_sa(mp, ira, &esph);
2931                        if (mp == NULL)
2932                                return;

2934                        ASSERT(esph != NULL);
2935                        ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
2936                        ASSERT(ira->ira_ipsec_esp_sa != NULL);
2937                        ASSERT(ira->ira_ipsec_esp_sa->ipsa_input_func != NULL);

2939                        esp_in_udp_sa = ((ira->ira_ipsec_esp_sa->ipsa_flags &
2940                            IPSA_F_NATT) != 0);
2941                        esp_in_udp_packet =
2942                            (ira->ira_flags & IRAF_ESP_UDP_PORTS) != 0;

2944                        /*
2945                         * The following is a fancy, but quick, way of saying:
2946                         * ESP-in-UDP SA and Raw ESP packet --> drop
2947                         *    OR
2948                         * ESP SA and ESP-in-UDP packet --> drop
2949                         */
2950                        if (esp_in_udp_sa != esp_in_udp_packet) {
2951                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2952                                ip_drop_packet(mp, B_TRUE, ira->ira_ill,
2953                                    DROPPER(ipss, ipds_esp_no_sa),
2954                                    &ipss->ipsec_dropper);
2955                                return;
2956                        }
2957                        mp = ira->ira_ipsec_esp_sa->ipsa_input_func(mp, esph,
2958                            ira);
2959                } else {
2960                        ah_t *ah;

2962                        mp = ipsec_inbound_ah_sa(mp, ira, &ah);
2963                        if (mp == NULL)
2964                                return;
```



```
2966                        ASSERT(ah != NULL);
2967                        ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
2968                        ASSERT(ira->ira_ipsec_ah_sa != NULL);
2969                        ASSERT(ira->ira_ipsec_ah_sa->ipsa_input_func != NULL);
2970                        mp = ira->ira_ipsec_ah_sa->ipsa_input_func(mp, ah,
2971                            ira);
2972                }

2974                if (mp == NULL) {
2975                        /*
2976                         * Either it failed or is pending. In the former case
2977                         * ipIfStatsInDiscards was increased.
2978                         */
2979                        return;
2980                }
2981                /* we're done with IPsec processing, send it up */
2982                ip_input_post_ipsec(mp, ira);
2983                return;
2984        }
2985        case IPPROTO_ENCAP: {
2986                ipha_t          *inner_ipha;

2988                /*
2989                 * Handle self-encapsulated packets (IP-in-IP where
2990                 * the inner addresses == the outer addresses).
2991                 */
2992                if ((uchar_t *)ipha + ip_hdr_length + sizeof (ipha_t) >
2993                    mp->b_wptr) {
2994                        if (ira->ira_pktlen <
2995                            ip_hdr_length + sizeof (ipha_t)) {
2996                                BUMP_MIB(ill->ill_ip_mib,
2997                                    ipIfStatsInTruncatedPkts);
2998                                ip_drop_input("ipIfStatsInTruncatedPkts",
2999                                    mp, ill);
3000                                freemsg(mp);
3001                                return;
3002                        }
3003                        ipha = ip_pullup(mp, (uchar_t *)ipha + ip_hdr_length +
3004                            sizeof (ipha_t) - mp->b_rptr, ira);
3005                        if (ipha == NULL) {
3006                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3007                                ip_drop_input("ipIfStatsInDiscards", mp, ill);
3008                                freemsg(mp);
3009                                return;
3010                        }
3011                }
3012                inner_ipha = (ipha_t *)((uchar_t *)ipha + ip_hdr_length);
3013                /*
3014                 * Check the sanity of the inner IP header.
3015                 */
3016                if ((IPH_HDR_VERSION(inner_ipha) != IPV4_VERSION)) {
3017                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3018                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
3019                        freemsg(mp);
3020                        return;
3021                }
3022                if (IPH_HDR_LENGTH(inner_ipha) < sizeof (ipha_t)) {
3023                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3024                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
3025                        freemsg(mp);
3026                        return;
3027                }
3028                if (inner_ipha->ipha_src != ipha->ipha_src ||
3029                    inner_ipha->ipha_dst != ipha->ipha_dst) {
3030                        /* We fallthru to iptun fanout below */
3031                        goto iptun;
```

```
3032                }

3034                /*
3035                 * Self-encapsulated tunnel packet. Remove
3036                 * the outer IP header and fanout again.
3037                 * We also need to make sure that the inner
3038                 * header is pulled up until options.
3039                 */
3040                mp->b_rptr = (uchar_t *)inner_ipha;
3041                ipha = inner_ipha;
3042                ip_hdr_length = IPH_HDR_LENGTH(ipha);
3043                if ((uchar_t *)ipha + ip_hdr_length > mp->b_wptr) {
3044                        if (ira->ira_pktlen <
3045                            (uchar_t *)ipha + ip_hdr_length - mp->b_rptr) {
3046                                BUMP_MIB(ill->ill_ip_mib,
3047                                    ipIfStatsInTruncatedPkts);
3048                                ip_drop_input("ipIfStatsInTruncatedPkts",
3049                                    mp, ill);
3050                                freemsg(mp);
3051                                return;
3052                        }
3053                        ipha = ip_pullup(mp,
3054                            (uchar_t *)ipha + ip_hdr_length - mp->b_rptr, ira);
3055                        if (ipha == NULL) {
3056                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3057                                ip_drop_input("ipIfStatsInDiscards", mp, ill);
3058                                freemsg(mp);
3059                                return;
3060                        }
3061                }
3062                if (ip_hdr_length > sizeof (ipha_t)) {
3063                        /* We got options on the inner packet. */
3064                        ipaddr_t        dst = ipha->ipha_dst;
3065                        int             error = 0;

3067                        dst = ip_input_options(ipha, dst, mp, ira, &error);
3068                        if (error != 0) {
3069                                /*
3070                                 * An ICMP error has been sent and the packet
3071                                 * has been dropped.
3072                                 */
3073                                return;
3074                        }
3075                        if (dst != ipha->ipha_dst) {
3076                                /*
3077                                 * Someone put a source-route in
3078                                 * the inside header of a self-
3079                                 * encapsulated packet.  Drop it
3080                                 * with extreme prejudice and let
3081                                 * the sender know.
3082                                 */
3083                                ip_drop_input("ICMP_SOURCE_ROUTE_FAILED",
3084                                    mp, ill);
3085                                icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED,
3086                                    ira);
3087                                return;
3088                        }
3089                }
3090                if (!(ira->ira_flags & IRAF_IPSEC_SECURE)) {
3091                        /*
3092                         * This means that somebody is sending
3093                         * Self-encapsualted packets without AH/ESP.
3094                         *
3095                         * Send this packet to find a tunnel endpoint.
3096                         * if I can't find one, an ICMP
3097                         * PROTOCOL_UNREACHABLE will get sent.
```



```
3098                         */
3099                        protocol = ipha->ipha_protocol;
3100                        ira->ira_protocol = protocol;
3101                        goto iptun;
3102                }

3104                /* Update based on removed IP header */
3105                ira->ira_ip_hdr_length = ip_hdr_length;
3106                ira->ira_pktlen = ntohs(ipha->ipha_length);

3108                if (ira->ira_flags & IRAF_IPSEC_DECAPS) {
3109                        /*
3110                         * This packet is self-encapsulated multiple
3111                         * times. We don't want to recurse infinitely.
3112                         * To keep it simple, drop the packet.
3113                         */
3114                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3115                        ip_drop_input("ipIfStatsInDiscards", mp, ill);
3116                        freemsg(mp);
3117                        return;
3118                }
3119                ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
3120                ira->ira_flags |= IRAF_IPSEC_DECAPS;

3122                ip_input_post_ipsec(mp, ira);
3123                return;
3124        }

3126        iptun:  /* IPPROTO_ENCAPS that is not self-encapsulated */
3127        case IPPROTO_IPV6:
3128                /* iptun will verify trusted label */
3129                connp = ipcl_classify_v4(mp, protocol, ip_hdr_length,
3130                    ira, ipst);
3131                if (connp != NULL) {
3132                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
3133                        ira->ira_ill = ira->ira_rill = NULL;
3134                        (connp->conn_recv)(connp, mp, NULL, ira);
3135                        CONN_DEC_REF(connp);
3136                        ira->ira_ill = ill;
3137                        ira->ira_rill = rill;
3138                        return;
3139                }
3140                /* FALLTHRU */
3141        default:
3142                /*
3143                 * On a labeled system, we have to check whether the zone
3144                 * itself is permitted to receive raw traffic.
3145                 */
3146                if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
3147                        if (!tsol_can_accept_raw(mp, ira, B_FALSE)) {
3148                                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3149                                ip_drop_input("ipIfStatsInDiscards", mp, ill);
3150                                freemsg(mp);
3151                                return;
3152                        }
3153                }
3154                break;
3155        }

3157        /*
3158         * The above input functions may have returned the pulled up message.
3159         * So ipha need to be reinitialized.
3160         */
3161        ipha = (ipha_t *)mp->b_rptr;
3162        ira->ira_protocol = protocol = ipha->ipha_protocol;
3163        if (ipst->ips_ipcl_proto_fanout_v4[protocol].connf_head == NULL) {
```

```
3164                 /*
3165                  * No user-level listener for these packets packets.
3166                  * Check for IPPROTO_ENCAP...
3167                  */
3168                 if (protocol == IPPROTO_ENCAP && ipst->ips_ip_g_mrouter) {
3169                         /*
3170                          * Check policy here,
3171                          * THEN ship off to ip_mroute_decap().
3172                          *
3173                          * BTW,  If I match a configured IP-in-IP
3174                          * tunnel above, this path will not be reached, and
3175                          * ip_mroute_decap will never be called.
3176                          */
3177                         mp = ipsec_check_global_policy(mp, connp,
3178                             ipha, NULL, ira, ns);
3179                         if (mp != NULL) {
3180                                 ip_mroute_decap(mp, ira);
3181                         } /* Else we already freed everything! */
3182                 } else {
3183                         ip_proto_not_sup(mp, ira);
3184                 }
3185                 return;
3186         }

3188         /*
3189          * Handle fanout to raw sockets.  There
3190          * can be more than one stream bound to a particular
3191          * protocol.  When this is the case, each one gets a copy
3192          * of any incoming packets.
3193          */
3194         ASSERT(ira->ira_protocol == ipha->ipha_protocol);
3195         ip_fanout_proto_v4(mp, ipha, ira);
3196         return;

3198 discard:
3199         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3200         ip_drop_input("ipIfStatsInDiscards", mp, ill);
3201         freemsg(mp);
3202 #undef rptr
3203 }
```

```
**********************************************************
   74252 Mon Jul  9 14:38:17 2012
new/usr/src/uts/common/inet/ip/ip_output.c
dccp: reset packet
**********************************************************
______unchanged_portion_omitted_

1606 /*
1607  * Calculate a checksum ignoring any hardware capabilities
1608  *
1609  * Returns B_FALSE if the packet was too short for the checksum. Caller
1610  * should free and do stats.
1611  */
1612 static boolean_t
1613 ip_output_sw_cksum_v4(mblk_t *mp, ipha_t *ipha, ip_xmit_attr_t *ixa)
1614 {
1615         ip_stack_t      *ipst = ixa->ixa_ipst;
1616         uint_t          pktlen = ixa->ixa_pktlen;
1617         uint16_t        *cksump;
1618         uint32_t        cksum;
1619         uint8_t         protocol = ixa->ixa_protocol;
1620         uint16_t        ip_hdr_length = ixa->ixa_ip_hdr_length;
1621         ipaddr_t        dst = ipha->ipha_dst;
1622         ipaddr_t        src = ipha->ipha_src;

1624         /* Just in case it contained garbage */
1625         DB_CKSUMFLAGS(mp) &= ~HCK_FLAGS;

1627         /*
1628          * Calculate ULP checksum
1629          */
1630         if (protocol == IPPROTO_TCP) {
1631                 cksump = IPH_TCPH_CHECKSUMP(ipha, ip_hdr_length);
1632                 cksum = IP_TCP_CSUM_COMP;
1633         } else if (protocol == IPPROTO_UDP) {
1634                 cksump = IPH_UDPH_CHECKSUMP(ipha, ip_hdr_length);
1635                 cksum = IP_UDP_CSUM_COMP;
1636         } else if (protocol == IPPROTO_SCTP) {
1637                 sctp_hdr_t      *sctph;

1639                 ASSERT(MBLKL(mp) >= (ip_hdr_length + sizeof (*sctph)));
1640                 sctph = (sctp_hdr_t *)(mp->b_rptr + ip_hdr_length);
1641                 /*
1642                  * Zero out the checksum field to ensure proper
1643                  * checksum calculation.
1644                  */
1645                 sctph->sh_chksum = 0;
1646 #ifdef  DEBUG
1647                 if (!skip_sctp_cksum)
1648 #endif
1649                         sctph->sh_chksum = sctp_cksum(mp, ip_hdr_length);
1650                 goto ip_hdr_cksum;
1651         } else if (protocol == IPPROTO_DCCP) {
1652                 cksump = IPH_DCCPH_CHECKSUMP(ipha, ip_hdr_length);
1653                 cksum = IP_DCCP_CSUM_COMP;
1654 #endif /* ! codereview */
1655         } else {
1656                 goto ip_hdr_cksum;
1657         }

1659         /* ULP puts the checksum field is in the first mblk */
1660         ASSERT(((uchar_t *)cksump) + sizeof (uint16_t) <= mp->b_wptr);

1662         /*
1663          * We accumulate the pseudo header checksum in cksum.
1664          * This is pretty hairy code, so watch close.  One
```



```
1665          * thing to keep in mind is that UDP and TCP have
1666          * stored their respective datagram lengths in their
1667          * checksum fields.  This lines things up real nice.
1668          */
1669         cksum += (dst >> 16) + (dst & 0xFFFF) + (src >> 16) + (src & 0xFFFF);

1671         cksum = IP_CSUM(mp, ip_hdr_length, cksum);
1672         /*
1673          * For UDP/IPv4 a zero means that the packets wasn't checksummed.
1674          * Change to 0xffff
1675          */
1676         if (protocol == IPPROTO_UDP && cksum == 0)
1677                 *cksump = ~cksum;
1678         else
1679                 *cksump = cksum;

1681         IP_STAT(ipst, ip_out_sw_cksum);
1682         IP_STAT_UPDATE(ipst, ip_out_sw_cksum_bytes, pktlen);

1684 ip_hdr_cksum:
1685         /* Calculate IPv4 header checksum */
1686         ipha->ipha_hdr_checksum = 0;
1687         ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
1688         return (B_TRUE);
1689 }

1691 /*
1692  * Calculate the ULP checksum - try to use hardware.
1693  * In the case of MULTIRT, broadcast or multicast the
1694  * IXAF_NO_HW_CKSUM is set in which case we use software.
1695  *
1696  * If the hardware supports IP header checksum offload; then clear the
1697  * contents of IP header checksum field as expected by NIC.
1698  * Do this only if we offloaded either full or partial sum.
1699  *
1700  * Returns B_FALSE if the packet was too short for the checksum. Caller
1701  * should free and do stats.
1702  */
1703 static boolean_t
1704 ip_output_cksum_v4(iaflags_t ixaflags, mblk_t *mp, ipha_t *ipha,
1705     ip_xmit_attr_t *ixa, ill_t *ill)
1706 {
1707         uint_t          pktlen = ixa->ixa_pktlen;
1708         uint16_t        *cksump;
1709         uint16_t        hck_flags;
1710         uint32_t        cksum;
1711         uint8_t         protocol = ixa->ixa_protocol;
1712         uint16_t        ip_hdr_length = ixa->ixa_ip_hdr_length;

1714         if ((ixaflags & IXAF_NO_HW_CKSUM) || !ILL_HCKSUM_CAPABLE(ill) ||
1715             !dohwcksum) {
1716                 return (ip_output_sw_cksum_v4(mp, ipha, ixa));
1717         }

1719         /*
1720          * Calculate ULP checksum. Note that we don't use cksump and cksum
1721          * if the ill has FULL support.
1722          */
1723         if (protocol == IPPROTO_TCP) {
1724                 cksump = IPH_TCPH_CHECKSUMP(ipha, ip_hdr_length);
1725                 cksum = IP_TCP_CSUM_COMP;       /* Pseudo-header cksum */
1726         } else if (protocol == IPPROTO_UDP) {
1727                 cksump = IPH_UDPH_CHECKSUMP(ipha, ip_hdr_length);
1728                 cksum = IP_UDP_CSUM_COMP;       /* Pseudo-header cksum */
1729         } else if (protocol == IPPROTO_SCTP) {
1730                 sctp_hdr_t      *sctph;
```

```
1732                ASSERT(MBLKL(mp) >= (ip_hdr_length + sizeof (*sctph)));
1733                sctph = (sctp_hdr_t *)(mp->b_rptr + ip_hdr_length);
1734                /*
1735                 * Zero out the checksum field to ensure proper
1736                 * checksum calculation.
1737                 */
1738                sctph->sh_chksum = 0;
1739 #ifdef  DEBUG
1740                if (!skip_sctp_cksum)
1741 #endif
1742                        sctph->sh_chksum = sctp_cksum(mp, ip_hdr_length);
1743                goto ip_hdr_cksum;
1744        } else if (protocol == IPPROTO_DCCP) {
1745                cksump = IPH_DCCPH_CHECKSUMP(ipha, ip_hdr_length);
1746                cksum = IP_DCCP_CSUM_COMP;
1747 #endif /* ! codereview */
1748        } else {
1749        ip_hdr_cksum:
1750                /* Calculate IPv4 header checksum */
1751                ipha->ipha_hdr_checksum = 0;
1752                ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
1753                return (B_TRUE);
1754        }

1756        /* ULP puts the checksum field is in the first mblk */
1757        ASSERT(((uchar_t *)cksump) + sizeof (uint16_t) <= mp->b_wptr);

1759        /*
1760         * Underlying interface supports hardware checksum offload for
1761         * the payload; leave the payload checksum for the hardware to
1762         * calculate.  N.B: We only need to set up checksum info on the
1763         * first mblk.
1764         */
1765        hck_flags = ill->ill_hcksum_capab->ill_hcksum_txflags;

1767        DB_CKSUMFLAGS(mp) &= ~HCK_FLAGS;
1768        if (hck_flags & HCKSUM_INET_FULL_V4) {
1769                /*
1770                 * Hardware calculates pseudo-header, header and the
1771                 * payload checksums, so clear the checksum field in
1772                 * the protocol header.
1773                 */
1774                *cksump = 0;
1775                DB_CKSUMFLAGS(mp) |= HCK_FULLCKSUM;

1777                ipha->ipha_hdr_checksum = 0;
1778                if (hck_flags & HCKSUM_IPHDRCKSUM) {
1779                        DB_CKSUMFLAGS(mp) |= HCK_IPV4_HDRCKSUM;
1780                } else {
1781                        ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
1782                }
1783                return (B_TRUE);
1784        }
1785        if ((hck_flags) & HCKSUM_INET_PARTIAL)  {
1786                ipaddr_t        dst = ipha->ipha_dst;
1787                ipaddr_t        src = ipha->ipha_src;
1788                /*
1789                 * Partial checksum offload has been enabled.  Fill
1790                 * the checksum field in the protocol header with the
1791                 * pseudo-header checksum value.
1792                 *
1793                 * We accumulate the pseudo header checksum in cksum.
1794                 * This is pretty hairy code, so watch close.  One
1795                 * thing to keep in mind is that UDP and TCP have
1796                 * stored their respective datagram lengths in their
```



```
1797                 * checksum fields.  This lines things up real nice.
1798                 */
1799                cksum += (dst >> 16) + (dst & 0xFFFF) +
1800                    (src >> 16) + (src & 0xFFFF);
1801                cksum += *(cksump);
1802                cksum = (cksum & 0xFFFF) + (cksum >> 16);
1803                *(cksump) = (cksum & 0xFFFF) + (cksum >> 16);

1805                /*
1806                 * Offsets are relative to beginning of IP header.
1807                 */
1808                DB_CKSUMSTART(mp) = ip_hdr_length;
1809                DB_CKSUMSTUFF(mp) = (uint8_t *)cksump - (uint8_t *)ipha;
1810                DB_CKSUMEND(mp) = pktlen;
1811                DB_CKSUMFLAGS(mp) |= HCK_PARTIALCKSUM;

1813                ipha->ipha_hdr_checksum = 0;
1814                if (hck_flags & HCKSUM_IPHDRCKSUM) {
1815                        DB_CKSUMFLAGS(mp) |= HCK_IPV4_HDRCKSUM;
1816                } else {
1817                        ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
1818                }
1819                return (B_TRUE);
1820        }
1821        /* Hardware capabilities include neither full nor partial IPv4 */
1822        return (ip_output_sw_cksum_v4(mp, ipha, ixa));
1823 }

1825 /*
1826  * ire_sendfn for offlink and onlink destinations.
1827  * Also called from the multicast, broadcast, multirt send functions.
1828  *
1829  * Assumes that the caller has a hold on the ire.
1830  *
1831  * This function doesn't care if the IRE just became condemned since that
1832  * can happen at any time.
1833  */
1834 /* ARGSUSED */
1835 int
1836 ire_send_wire_v4(ire_t *ire, mblk_t *mp, void *iph_arg,
1837     ip_xmit_attr_t *ixa, uint32_t *identp)
1838 {
1839        ip_stack_t      *ipst = ixa->ixa_ipst;
1840        ipha_t          *ipha = (ipha_t *)iph_arg;
1841        iaflags_t       ixaflags = ixa->ixa_flags;
1842        ill_t           *ill;

1844        ASSERT(ixa->ixa_nce != NULL);
1845        ill = ixa->ixa_nce->nce_ill;

1847        if (ixaflags & IXAF_DONTROUTE)
1848                ipha->ipha_ttl = 1;

1850        /*
1851         * Assign an ident value for this packet. There could be other
1852         * threads targeting the same destination, so we have to arrange
1853         * for a atomic increment.  Note that we use a 32-bit atomic add
1854         * because it has better performance than its 16-bit sibling.
1855         *
1856         * Normally ixa_extra_ident is 0, but in the case of LSO it will
1857         * be the number of TCP segments  that the driver/hardware will
1858         * extraly construct.
1859         *
1860         * If running in cluster mode and if the source address
1861         * belongs to a replicated service then vector through
1862         * cl_inet_ipident vector to allocate ip identifier
```

```
1863          * NOTE: This is a contract private interface with the
1864          * clustering group.
1865          */
1866         if (cl_inet_ipident != NULL) {
1867                 ipaddr_t src = ipha->ipha_src;
1868                 ipaddr_t dst = ipha->ipha_dst;
1869                 netstackid_t stack_id = ipst->ips_netstack->netstack_stackid;

1871                 ASSERT(cl_inet_isclusterwide != NULL);
1872                 if ((*cl_inet_isclusterwide)(stack_id, IPPROTO_IP,
1873                     AF_INET, (uint8_t *)(uintptr_t)src, NULL)) {
1874                         /*
1875                          * Note: not correct with LSO since we can't allocate
1876                          * ixa_extra_ident+1 consecutive values.
1877                          */
1878                         ipha->ipha_ident = (*cl_inet_ipident)(stack_id,
1879                             IPPROTO_IP, AF_INET, (uint8_t *)(uintptr_t)src,
1880                             (uint8_t *)(uintptr_t)dst, NULL);
1881                 } else {
1882                         ipha->ipha_ident = atomic_add_32_nv(identp,
1883                             ixa->ixa_extra_ident + 1);
1884                 }
1885         } else {
1886                 ipha->ipha_ident = atomic_add_32_nv(identp,
1887                     ixa->ixa_extra_ident + 1);
1888         }
1889 #ifndef _BIG_ENDIAN
1890         ipha->ipha_ident = htons(ipha->ipha_ident);
1891 #endif

1893         /*
1894          * This might set b_band, thus the IPsec and fragmentation
1895          * code in IP ensures that b_band is updated in the first mblk.
1896          */
1897         if (IPP_ENABLED(IPP_LOCAL_OUT, ipst)) {
1898                 /* ip_process translates an IS_UNDER_IPMP */
1899                 mp = ip_process(IPP_LOCAL_OUT, mp, ill, ill);
1900                 if (mp == NULL) {
1901                         /* ip_drop_packet and MIB done */
1902                         return (0);     /* Might just be delayed */
1903                 }
1904         }

1906         /*
1907          * Verify any IPv4 options.
1908          *
1909          * The presense of IP options also forces the network stack to
1910          * calculate the checksum in software.  This is because:
1911          *
1912          * Wrap around: certain partial-checksum NICs (eri, ce) limit
1913          * the size of "start offset" width to 6-bit.  This effectively
1914          * sets the largest value of the offset to 64-bytes, starting
1915          * from the MAC header.  When the cumulative MAC and IP headers
1916          * exceed such limit, the offset will wrap around.  This causes
1917          * the checksum to be calculated at the wrong place.
1918          *
1919          * IPv4 source routing: none of the full-checksum capable NICs
1920          * is capable of correctly handling the IPv4 source-routing
1921          * option for purposes of calculating the pseudo-header; the
1922          * actual destination is different from the destination in the
1923          * header which is that of the next-hop.  (This case may not be
1924          * true for NICs which can parse IPv6 extension headers, but
1925          * we choose to simplify the implementation by not offloading
1926          * checksum when they are present.)
1927          */
1928         if (!IS_SIMPLE_IPH(ipha)) {
```



```
1929                 ixaflags = ixa->ixa_flags |= IXAF_NO_HW_CKSUM;
1930                 /* An IS_UNDER_IPMP ill is ok here */
1931                 if (ip_output_options(mp, ipha, ixa, ill)) {
1932                         /* Packet has been consumed and ICMP error sent */
1933                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
1934                         return (EINVAL);
1935                 }
1936         }

1938         /*
1939          * To handle IPsec/iptun's labeling needs we need to tag packets
1940          * while we still have ixa_tsl
1941          */
1942         if (is_system_labeled() && ixa->ixa_tsl != NULL &&
1943             (ill->ill_mactype == DL_6TO4 || ill->ill_mactype == DL_IPV4 ||
1944             ill->ill_mactype == DL_IPV6)) {
1945                 cred_t *newcr;

1947                 newcr = copycred_from_tslabel(ixa->ixa_cred, ixa->ixa_tsl,
1948                     KM_NOSLEEP);
1949                 if (newcr == NULL) {
1950                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
1951                         ip_drop_output("ipIfStatsOutDiscards - newcr",
1952                             mp, ill);
1953                         freemsg(mp);
1954                         return (ENOBUFS);
1955                 }
1956                 mblk_setcred(mp, newcr, NOPID);
1957                 crfree(newcr);  /* mblk_setcred did its own crhold */
1958         }

1960         if (ixa->ixa_pktlen > ixa->ixa_fragsize ||
1961             (ixaflags & IXAF_IPSEC_SECURE)) {
1962                 uint32_t pktlen;

1964                 pktlen = ixa->ixa_pktlen;
1965                 if (ixaflags & IXAF_IPSEC_SECURE)
1966                         pktlen += ipsec_out_extra_length(ixa);

1968                 if (pktlen > IP_MAXPACKET)
1969                         return (EMSGSIZE);

1971                 if (ixaflags & IXAF_SET_ULP_CKSUM) {
1972                         /*
1973                          * Compute ULP checksum and IP header checksum
1974                          * using software
1975                          */
1976                         if (!ip_output_sw_cksum_v4(mp, ipha, ixa)) {
1977                                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
1978                                 ip_drop_output("ipIfStatsOutDiscards", mp, ill);
1979                                 freemsg(mp);
1980                                 return (EINVAL);
1981                         }
1982                 } else {
1983                         /* Calculate IPv4 header checksum */
1984                         ipha->ipha_hdr_checksum = 0;
1985                         ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
1986                 }

1988                 /*
1989                  * If this packet would generate a icmp_frag_needed
1990                  * message, we need to handle it before we do the IPsec
1991                  * processing. Otherwise, we need to strip the IPsec
1992                  * headers before we send up the message to the ULPs
1993                  * which becomes messy and difficult.
1994                  *
```

```
1995                 * We check using IXAF_DONTFRAG. The DF bit in the header
1996                 * is not inspected - it will be copied to any generated
1997                 * fragments.
1998                 */
1999                if ((pktlen > ixa->ixa_fragsize) &&
2000                    (ixaflags & IXAF_DONTFRAG)) {
2001                        /* Generate ICMP and return error */
2002                        ip_recv_attr_t  iras;

2004                        DTRACE_PROBE4(ip4__fragsize__fail, uint_t, pktlen,
2005                            uint_t, ixa->ixa_fragsize, uint_t, ixa->ixa_pktlen,
2006                            uint_t, ixa->ixa_pmtu);

2008                        bzero(&iras, sizeof (iras));
2009                        /* Map ixa to ira including IPsec policies */
2010                        ipsec_out_to_in(ixa, ill, &iras);

2012                        ip_drop_output("ICMP_FRAG_NEEDED", mp, ill);
2013                        icmp_frag_needed(mp, ixa->ixa_fragsize, &iras);
2014                        /* We moved any IPsec refs from ixa to iras */
2015                        ira_cleanup(&iras, B_FALSE);
2016                        return (EMSGSIZE);
2017                }
2018                DTRACE_PROBE4(ip4__fragsize__ok, uint_t, pktlen,
2019                    uint_t, ixa->ixa_fragsize, uint_t, ixa->ixa_pktlen,
2020                    uint_t, ixa->ixa_pmtu);

2022                if (ixaflags & IXAF_IPSEC_SECURE) {
2023                        /*
2024                         * Pass in sufficient information so that
2025                         * IPsec can determine whether to fragment, and
2026                         * which function to call after fragmentation.
2027                         */
2028                        return (ipsec_out_process(mp, ixa));
2029                }
2030                return (ip_fragment_v4(mp, ixa->ixa_nce, ixaflags,
2031                    ixa->ixa_pktlen, ixa->ixa_fragsize, ixa->ixa_xmit_hint,
2032                    ixa->ixa_zoneid, ixa->ixa_no_loop_zoneid,
2033                    ixa->ixa_postfragfn, &ixa->ixa_cookie));
2034        }
2035        if (ixaflags & IXAF_SET_ULP_CKSUM) {
2036                /* Compute ULP checksum and IP header checksum */
2037                /* An IS_UNDER_IPMP ill is ok here */
2038                if (!ip_output_cksum_v4(ixaflags, mp, ipha, ixa, ill)) {
2039                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2040                        ip_drop_output("ipIfStatsOutDiscards", mp, ill);
2041                        freemsg(mp);
2042                        return (EINVAL);
2043                }
2044        } else {
2045                /* Calculate IPv4 header checksum */
2046                ipha->ipha_hdr_checksum = 0;
2047                ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
2048        }
2049        return ((ixa->ixa_postfragfn)(mp, ixa->ixa_nce, ixaflags,
2050            ixa->ixa_pktlen, ixa->ixa_xmit_hint, ixa->ixa_zoneid,
2051            ixa->ixa_no_loop_zoneid, &ixa->ixa_cookie));
2052 }

2054 /*
2055  * Send mp into ip_input
2056  * Common for IPv4 and IPv6
2057  */
2058 void
2059 ip_postfrag_loopback(mblk_t *mp, nce_t *nce, iaflags_t ixaflags,
2060     uint_t pkt_len, zoneid_t nolzid)
```



```
2061 {
2062        rtc_t           rtc;
2063        ill_t           *ill = nce->nce_ill;
2064        ip_recv_attr_t  iras;   /* NOTE: No bzero for performance */
2065        ncec_t          *ncec;

2067        ncec = nce->nce_common;
2068        iras.ira_flags = IRAF_VERIFY_IP_CKSUM | IRAF_VERIFY_ULP_CKSUM |
2069            IRAF_LOOPBACK | IRAF_L2SRC_LOOPBACK;
2070        if (ncec->ncec_flags & NCE_F_BCAST)
2071                iras.ira_flags |= IRAF_L2DST_BROADCAST;
2072        else if (ncec->ncec_flags & NCE_F_MCAST)
2073                iras.ira_flags |= IRAF_L2DST_MULTICAST;

2075        iras.ira_free_flags = 0;
2076        iras.ira_cred = NULL;
2077        iras.ira_cpid = NOPID;
2078        iras.ira_tsl = NULL;
2079        iras.ira_zoneid = ALL_ZONES;
2080        iras.ira_pktlen = pkt_len;
2081        UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCInOctets, iras.ira_pktlen);
2082        BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInReceives);

2084        if (ixaflags & IXAF_IS_IPV4)
2085                iras.ira_flags |= IRAF_IS_IPV4;

2087        iras.ira_ill = iras.ira_rill = ill;
2088        iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
2089        iras.ira_rifindex = iras.ira_ruifindex;
2090        iras.ira_mhip = NULL;

2092        iras.ira_flags |= ixaflags & IAF_MASK;
2093        iras.ira_no_loop_zoneid = nolzid;

2095        /* Broadcast and multicast doesn't care about the squeue */
2096        iras.ira_sqp = NULL;

2098        rtc.rtc_ire = NULL;
2099        if (ixaflags & IXAF_IS_IPV4) {
2100                ipha_t          *ipha = (ipha_t *)mp->b_rptr;

2102                rtc.rtc_ipaddr = INADDR_ANY;

2104                (*ill->ill_inputfn)(mp, ipha, &ipha->ipha_dst, &iras, &rtc);
2105                if (rtc.rtc_ire != NULL) {
2106                        ASSERT(rtc.rtc_ipaddr != INADDR_ANY);
2107                        ire_refrele(rtc.rtc_ire);
2108                }
2109        } else {
2110                ip6_t           *ip6h = (ip6_t *)mp->b_rptr;

2112                rtc.rtc_ip6addr = ipv6_all_zeros;

2114                (*ill->ill_inputfn)(mp, ip6h, &ip6h->ip6_dst, &iras, &rtc);
2115                if (rtc.rtc_ire != NULL) {
2116                        ASSERT(!IN6_IS_ADDR_UNSPECIFIED(&rtc.rtc_ip6addr));
2117                        ire_refrele(rtc.rtc_ire);
2118                }
2119        }
2120        /* Any references to clean up? No hold on ira */
2121        if (iras.ira_flags & (IRAF_IPSEC_SECURE|IRAF_SYSTEM_LABELED))
2122                ira_cleanup(&iras, B_FALSE);
2123 }

2125 /*
2126  * Post fragmentation function for IRE_MULTICAST and IRE_BROADCAST which
```

```
 * looks at the IXAF_LOOPBACK_COPY flag.
 * Common for IPv4 and IPv6.
 *
 * If the loopback copy fails (due to no memory) but we send the packet out
 * on the wire we return no failure. Only in the case we supress the wire
 * sending do we take the loopback failure into account.
 *
 * Note that we do not perform DTRACE_IP7 and FW_HOOKS for the looped back copy.
 * Those operations are performed on this packet in ip_xmit() and it would
 * be odd to do it twice for the same packet.
 */
int
ip_postfrag_loopcheck(mblk_t *mp, nce_t *nce, iaflags_t ixaflags,
    uint_t pkt_len, uint32_t xmit_hint, zoneid_t szone, zoneid_t nolzid,
    uintptr_t *ixacookie)
{
        ill_t           *ill = nce->nce_ill;
        int             error = 0;

        /*
         * Check for IXAF_LOOPBACK_COPY - send a copy to ip as if the driver
         * had looped it back
         */
        if (ixaflags & IXAF_LOOPBACK_COPY) {
                mblk_t          *mp1;

                mp1 = copymsg(mp);
                if (mp1 == NULL) {
                        /* Failed to deliver the loopback copy. */
                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
                        ip_drop_output("ipIfStatsOutDiscards", mp, ill);
                        error = ENOBUFS;
                } else {
                        ip_postfrag_loopback(mp1, nce, ixaflags, pkt_len,
                            nolzid);
                }
        }

        /*
         * If TTL = 0 then only do the loopback to this host i.e. we are
         * done. We are also done if this was the
         * loopback interface since it is sufficient
         * to loopback one copy of a multicast packet.
         */
        if (ixaflags & IXAF_IS_IPV4) {
                ipha_t *ipha = (ipha_t *)mp->b_rptr;

                if (ipha->ipha_ttl == 0) {
                        ip_drop_output("multicast ipha_ttl not sent to wire",
                            mp, ill);
                        freemsg(mp);
                        return (error);
                }
        } else {
                ip6_t   *ip6h = (ip6_t *)mp->b_rptr;

                if (ip6h->ip6_hops == 0) {
                        ip_drop_output("multicast ipha_ttl not sent to wire",
                            mp, ill);
                        freemsg(mp);
                        return (error);
                }
        }
        if (nce->nce_ill->ill_wq == NULL) {
                /* Loopback interface */
                ip_drop_output("multicast on lo0 not sent to wire", mp, ill);
```



```
                freemsg(mp);
                return (error);
        }

        return (ip_xmit(mp, nce, ixaflags, pkt_len, xmit_hint, szone, 0,
            ixacookie));
}

/*
 * Post fragmentation function for RTF_MULTIRT routes.
 * Since IRE_BROADCASTs can have RTF_MULTIRT, this function
 * checks IXAF_LOOPBACK_COPY.
 *
 * If no packet is sent due to failures then we return an errno, but if at
 * least one succeeded we return zero.
 */
int
ip_postfrag_multirt_v4(mblk_t *mp, nce_t *nce, iaflags_t ixaflags,
    uint_t pkt_len, uint32_t xmit_hint, zoneid_t szone, zoneid_t nolzid,
    uintptr_t *ixacookie)
{
        irb_t           *irb;
        ipha_t          *ipha = (ipha_t *)mp->b_rptr;
        ire_t           *ire;
        ire_t           *ire1;
        mblk_t          *mp1;
        nce_t           *nce1;
        ill_t           *ill = nce->nce_ill;
        ill_t           *ill1;
        ip_stack_t      *ipst = ill->ill_ipst;
        int             error = 0;
        int             num_sent = 0;
        int             err;
        uint_t          ire_type;
        ipaddr_t        nexthop;

        ASSERT(ixaflags & IXAF_IS_IPV4);

        /* Check for IXAF_LOOPBACK_COPY */
        if (ixaflags & IXAF_LOOPBACK_COPY) {
                mblk_t *mp1;

                mp1 = copymsg(mp);
                if (mp1 == NULL) {
                        /* Failed to deliver the loopback copy. */
                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
                        ip_drop_output("ipIfStatsOutDiscards", mp, ill);
                        error = ENOBUFS;
                } else {
                        ip_postfrag_loopback(mp1, nce, ixaflags, pkt_len,
                            nolzid);
                }
        }

        /*
         * Loop over RTF_MULTIRT for ipha_dst in the same bucket. Send
         * a copy to each one.
         * Use the nce (nexthop) and ipha_dst to find the ire.
         *
         * MULTIRT is not designed to work with shared-IP zones thus we don't
         * need to pass a zoneid or a label to the IRE lookup.
         */
        if (V4_PART_OF_V6(nce->nce_addr) == ipha->ipha_dst) {
                /* Broadcast and multicast case */
                ire = ire_ftable_lookup_v4(ipha->ipha_dst, 0, 0, 0,
                    NULL, ALL_ZONES, NULL, MATCH_IRE_DSTONLY, 0, ipst, NULL);
```

```
2259         } else {
2260                 ipaddr_t v4addr = V4_PART_OF_V6(nce->nce_addr);

2262                 /* Unicast case */
2263                 ire = ire_ftable_lookup_v4(ipha->ipha_dst, 0, v4addr, 0,
2264                     NULL, ALL_ZONES, NULL, MATCH_IRE_GW, 0, ipst, NULL);
2265         }

2267         if (ire == NULL ||
2268             (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) ||
2269             !(ire->ire_flags & RTF_MULTIRT)) {
2270                 /* Drop */
2271                 ip_drop_output("ip_postfrag_multirt didn't find route",
2272                     mp, nce->nce_ill);
2273                 if (ire != NULL)
2274                         ire_refrele(ire);
2275                 return (ENETUNREACH);
2276         }

2278         irb = ire->ire_bucket;
2279         irb_refhold(irb);
2280         for (ire1 = irb->irb_ire; ire1 != NULL; ire1 = ire1->ire_next) {
2281                 /*
2282                  * For broadcast we can have a mixture of IRE_BROADCAST and
2283                  * IRE_HOST due to the manually added IRE_HOSTs that are used
2284                  * to trigger the creation of the special CGTP broadcast routes.
2285                  * Thus we have to skip if ire_type doesn't match the original.
2286                  */
2287                 if (IRE_IS_CONDEMNED(ire1) ||
2288                     !(ire1->ire_flags & RTF_MULTIRT) ||
2289                     ire1->ire_type != ire->ire_type)
2290                         continue;

2292                 /* Do the ire argument one after the loop */
2293                 if (ire1 == ire)
2294                         continue;

2296                 ill1 = ire_nexthop_ill(ire1);
2297                 if (ill1 == NULL) {
2298                         /*
2299                          * This ire might not have been picked by
2300                          * ire_route_recursive, in which case ire_dep might
2301                          * not have been setup yet.
2302                          * We kick ire_route_recursive to try to resolve
2303                          * starting at ire1.
2304                          */
2305                         ire_t *ire2;
2306                         uint_t  match_flags = MATCH_IRE_DSTONLY;

2308                         if (ire1->ire_ill != NULL)
2309                                 match_flags |= MATCH_IRE_ILL;
2310                         ire2 = ire_route_recursive_impl_v4(ire1,
2311                             ire1->ire_addr, ire1->ire_type, ire1->ire_ill,
2312                             ire1->ire_zoneid, NULL, match_flags,
2313                             IRR_ALLOCATE, 0, ipst, NULL, NULL, NULL);
2314                         if (ire2 != NULL)
2315                                 ire_refrele(ire2);
2316                         ill1 = ire_nexthop_ill(ire1);
2317                 }

2319                 if (ill1 == NULL) {
2320                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2321                         ip_drop_output("ipIfStatsOutDiscards - no ill",
2322                             mp, ill);
2323                         error = ENETUNREACH;
2324                         continue;
```



```
2325                 }

2327                 /* Pick the addr and type to use for arp_nce_init */
2328                 if (nce->nce_common->ncec_flags & NCE_F_BCAST) {
2329                         ire_type = IRE_BROADCAST;
2330                         nexthop = ire1->ire_gateway_addr;
2331                 } else if (nce->nce_common->ncec_flags & NCE_F_MCAST) {
2332                         ire_type = IRE_MULTICAST;
2333                         nexthop = ipha->ipha_dst;
2334                 } else {
2335                         ire_type = ire1->ire_type;      /* Doesn't matter */
2336                         nexthop = ire1->ire_gateway_addr;
2337                 }

2339                 /* If IPMP meta or under, then we just drop */
2340                 if (ill1->ill_grp != NULL) {
2341                         BUMP_MIB(ill1->ill_ip_mib, ipIfStatsOutDiscards);
2342                         ip_drop_output("ipIfStatsOutDiscards - IPMP",
2343                             mp, ill1);
2344                         ill_refrele(ill1);
2345                         error = ENETUNREACH;
2346                         continue;
2347                 }

2349                 nce1 = arp_nce_init(ill1, nexthop, ire_type);
2350                 if (nce1 == NULL) {
2351                         BUMP_MIB(ill1->ill_ip_mib, ipIfStatsOutDiscards);
2352                         ip_drop_output("ipIfStatsOutDiscards - no nce",
2353                             mp, ill1);
2354                         ill_refrele(ill1);
2355                         error = ENETUNREACH;
2356                         continue;
2357                 }
2358                 mp1 = copymsg(mp);
2359                 if (mp1 == NULL) {
2360                         BUMP_MIB(ill1->ill_ip_mib, ipIfStatsOutDiscards);
2361                         ip_drop_output("ipIfStatsOutDiscards", mp, ill1);
2362                         nce_refrele(nce1);
2363                         ill_refrele(ill1);
2364                         error = ENOBUFS;
2365                         continue;
2366                 }
2367                 /* Preserve HW checksum for this copy */
2368                 DB_CKSUMSTART(mp1) = DB_CKSUMSTART(mp);
2369                 DB_CKSUMSTUFF(mp1) = DB_CKSUMSTUFF(mp);
2370                 DB_CKSUMEND(mp1) = DB_CKSUMEND(mp);
2371                 DB_CKSUMFLAGS(mp1) = DB_CKSUMFLAGS(mp);
2372                 DB_LSOMSS(mp1) = DB_LSOMSS(mp);

2374                 ire1->ire_ob_pkt_count++;
2375                 err = ip_xmit(mp1, nce1, ixaflags, pkt_len, xmit_hint, szone,
2376                     0, ixacookie);
2377                 if (err == 0)
2378                         num_sent++;
2379                 else
2380                         error = err;
2381                 nce_refrele(nce1);
2382                 ill_refrele(ill1);
2383         }
2384         irb_refrele(irb);
2385         ire_refrele(ire);
2386         /* Finally, the main one */
2387         err = ip_xmit(mp, nce, ixaflags, pkt_len, xmit_hint, szone, 0,
2388             ixacookie);
2389         if (err == 0)
2390                 num_sent++;
```

```
2391         else
2392                 error = err;
2393         if (num_sent > 0)
2394                 return (0);
2395         else
2396                 return (error);
2397 }

2399 /*
2400  * Verify local connectivity. This check is called by ULP fusion code.
2401  * The generation number on an IRE_LOCAL or IRE_LOOPBACK only changes if
2402  * the interface is brought down and back up. So we simply fail the local
2403  * process. The caller, TCP Fusion, should unfuse the connection.
2404  */
2405 boolean_t
2406 ip_output_verify_local(ip_xmit_attr_t *ixa)
2407 {
2408         ire_t           *ire = ixa->ixa_ire;

2410         if (!(ire->ire_type & (IRE_LOCAL | IRE_LOOPBACK)))
2411                 return (B_FALSE);

2413         return (ixa->ixa_ire->ire_generation == ixa->ixa_ire_generation);
2414 }

2416 /*
2417  * Local process for ULP loopback, TCP Fusion. Handle both IPv4 and IPv6.
2418  *
2419  * The caller must call ip_output_verify_local() first. This function handles
2420  * IPobs, FW_HOOKS, and/or IPsec cases sequentially.
2421  */
2422 mblk_t *
2423 ip_output_process_local(mblk_t *mp, ip_xmit_attr_t *ixa, boolean_t hooks_out,
2424     boolean_t hooks_in, conn_t *peer_connp)
2425 {
2426         ill_t           *ill = ixa->ixa_ire->ire_ill;
2427         ipha_t          *ipha = NULL;
2428         ip6_t           *ip6h = NULL;
2429         ip_stack_t      *ipst = ixa->ixa_ipst;
2430         iaflags_t       ixaflags = ixa->ixa_flags;
2431         ip_recv_attr_t  iras;
2432         int             error;

2434         ASSERT(mp != NULL);

2436         if (ixaflags & IXAF_IS_IPV4) {
2437                 ipha = (ipha_t *)mp->b_rptr;

2439                 /*
2440                  * If a callback is enabled then we need to know the
2441                  * source and destination zoneids for the packet. We already
2442                  * have those handy.
2443                  */
2444                 if (ipst->ips_ip4_observe.he_interested) {
2445                         zoneid_t szone, dzone;
2446                         zoneid_t stackzoneid;

2448                         stackzoneid = netstackid_to_zoneid(
2449                             ipst->ips_netstack->netstack_stackid);

2451                         if (stackzoneid == GLOBAL_ZONEID) {
2452                                 /* Shared-IP zone */
2453                                 dzone = ixa->ixa_ire->ire_zoneid;
2454                                 szone = ixa->ixa_zoneid;
2455                         } else {
2456                                 szone = dzone = stackzoneid;
```



```
2457                         }
2458                         ipobs_hook(mp, IPOBS_HOOK_LOCAL, szone, dzone, ill,
2459                             ipst);
2460                 }
2461                 DTRACE_IP7(send, mblk_t *, mp, conn_t *, NULL, void_ip_t *,
2462                     ipha, __dtrace_ipsr_ill_t *, ill, ipha_t *, ipha, ip6_t *,
2463                     NULL, int, 1);

2465                 /* FW_HOOKS: LOOPBACK_OUT */
2466                 if (hooks_out) {
2467                         DTRACE_PROBE4(ip4__loopback__out__start, ill_t *, NULL,
2468                             ill_t *, ill, ipha_t *, ipha, mblk_t *, mp);
2469                         FW_HOOKS(ipst->ips_ip4_loopback_out_event,
2470                             ipst->ips_ipv4firewall_loopback_out,
2471                             NULL, ill, ipha, mp, mp, 0, ipst, error);
2472                         DTRACE_PROBE1(ip4__loopback__out__end, mblk_t *, mp);
2473                 }
2474                 if (mp == NULL)
2475                         return (NULL);

2477                 /* FW_HOOKS: LOOPBACK_IN */
2478                 if (hooks_in) {
2479                         DTRACE_PROBE4(ip4__loopback__in__start, ill_t *, ill,
2480                             ill_t *, NULL, ipha_t *, ipha, mblk_t *, mp);
2481                         FW_HOOKS(ipst->ips_ip4_loopback_in_event,
2482                             ipst->ips_ipv4firewall_loopback_in,
2483                             ill, NULL, ipha, mp, mp, 0, ipst, error);
2484                         DTRACE_PROBE1(ip4__loopback__in__end, mblk_t *, mp);
2485                 }
2486                 if (mp == NULL)
2487                         return (NULL);

2489                 DTRACE_IP7(receive, mblk_t *, mp, conn_t *, NULL, void_ip_t *,
2490                     ipha, __dtrace_ipsr_ill_t *, ill, ipha_t *, ipha, ip6_t *,
2491                     NULL, int, 1);

2493                 /* Inbound IPsec polocies */
2494                 if (peer_connp != NULL) {
2495                         /* Map ixa to ira including IPsec policies. */
2496                         ipsec_out_to_in(ixa, ill, &iras);
2497                         mp = ipsec_check_inbound_policy(mp, peer_connp, ipha,
2498                             NULL, &iras);
2499                 }
2500         } else {
2501                 ip6h = (ip6_t *)mp->b_rptr;

2503                 /*
2504                  * If a callback is enabled then we need to know the
2505                  * source and destination zoneids for the packet. We already
2506                  * have those handy.
2507                  */
2508                 if (ipst->ips_ip6_observe.he_interested) {
2509                         zoneid_t szone, dzone;
2510                         zoneid_t stackzoneid;

2512                         stackzoneid = netstackid_to_zoneid(
2513                             ipst->ips_netstack->netstack_stackid);

2515                         if (stackzoneid == GLOBAL_ZONEID) {
2516                                 /* Shared-IP zone */
2517                                 dzone = ixa->ixa_ire->ire_zoneid;
2518                                 szone = ixa->ixa_zoneid;
2519                         } else {
2520                                 szone = dzone = stackzoneid;
2521                         }
2522                         ipobs_hook(mp, IPOBS_HOOK_LOCAL, szone, dzone, ill,
```

```
2523                            ipst);
2524                }
2525                DTRACE_IP7(send, mblk_t *, mp, conn_t *, NULL, void_ip_t *,
2526                    ip6h, __dtrace_ipsr_ill_t *, ill, ipha_t *, NULL, ip6_t *,
2527                    ip6h, int, 1);

2529                /* FW_HOOKS: LOOPBACK_OUT */
2530                if (hooks_out) {
2531                        DTRACE_PROBE4(ip6__loopback__out__start, ill_t *, NULL,
2532                            ill_t *, ill, ip6_t *, ip6h, mblk_t *, mp);
2533                        FW_HOOKS6(ipst->ips_ip6_loopback_out_event,
2534                            ipst->ips_ipv6firewall_loopback_out,
2535                            NULL, ill, ip6h, mp, mp, 0, ipst, error);
2536                        DTRACE_PROBE1(ip6__loopback__out__end, mblk_t *, mp);
2537                }
2538                if (mp == NULL)
2539                        return (NULL);

2541                /* FW_HOOKS: LOOPBACK_IN */
2542                if (hooks_in) {
2543                        DTRACE_PROBE4(ip6__loopback__in__start, ill_t *, ill,
2544                            ill_t *, NULL, ip6_t *, ip6h, mblk_t *, mp);
2545                        FW_HOOKS6(ipst->ips_ip6_loopback_in_event,
2546                            ipst->ips_ipv6firewall_loopback_in,
2547                            ill, NULL, ip6h, mp, mp, 0, ipst, error);
2548                        DTRACE_PROBE1(ip6__loopback__in__end, mblk_t *, mp);
2549                }
2550                if (mp == NULL)
2551                        return (NULL);

2553                DTRACE_IP7(receive, mblk_t *, mp, conn_t *, NULL, void_ip_t *,
2554                    ip6h, __dtrace_ipsr_ill_t *, ill, ipha_t *, NULL, ip6_t *,
2555                    ip6h, int, 1);

2557                /* Inbound IPsec polocies */
2558                if (peer_connp != NULL) {
2559                        /* Map ixa to ira including IPsec policies. */
2560                        ipsec_out_to_in(ixa, ill, &iras);
2561                        mp = ipsec_check_inbound_policy(mp, peer_connp, NULL,
2562                            ip6h, &iras);
2563                }
2564        }

2566        if (mp == NULL) {
2567                BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2568                ip_drop_input("ipIfStatsInDiscards", NULL, ill);
2569        }

2571        return (mp);
2572 }
```

```
**********************************************************
   84423 Mon Jul  9 14:38:18 2012
new/usr/src/uts/common/inet/ip/ipclassifier.c
dccp: conn_t
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * IP PACKET CLASSIFIER
 *
 * The IP packet classifier provides mapping between IP packets and persistent
 * connection state for connection-oriented protocols. It also provides
 * interface for managing connection states.
 *
 * The connection state is kept in conn_t data structure and contains, among
 * other things:
 *
 *      o local/remote address and ports
 *      o Transport protocol
 *      o squeue for the connection (for TCP only)
 *      o reference counter
 *      o Connection state
 *      o hash table linkage
 *      o interface/ire information
 *      o credentials
 *      o ipsec policy
 *      o send and receive functions.
 *      o mutex lock.
 *
 * Connections use a reference counting scheme. They are freed when the
 * reference counter drops to zero. A reference is incremented when connection
 * is placed in a list or table, when incoming packet for the connection arrives
 * and when connection is processed via squeue (squeue processing may be
 * asynchronous and the reference protects the connection from being destroyed
 * before its processing is finished).
 *
 * conn_recv is used to pass up packets to the ULP.
 * For TCP conn_recv changes. It is tcp_input_listener_unbound initially for
 * a listener, and changes to tcp_input_listener as the listener has picked a
 * good squeue. For other cases it is set to tcp_input_data.
 *
 * conn_recvicmp is used to pass up ICMP errors to the ULP.
 *
 * Classifier uses several hash tables:
```



```
 *
 *      ipcl_conn_fanout:       contains all TCP connections in CONNECTED state
 *      ipcl_bind_fanout:       contains all connections in BOUND state
 *      ipcl_proto_fanout:      IPv4 protocol fanout
 *      ipcl_proto_fanout_v6:   IPv6 protocol fanout
 *      ipcl_udp_fanout:        contains all UDP connections
 *      ipcl_iptun_fanout:      contains all IP tunnel connections
 *      ipcl_globalhash_fanout: contains all connections
 *
 * The ipcl_globalhash_fanout is used for any walkers (like snmp and Clustering)
 * which need to view all existing connections.
 *
 * All tables are protected by per-bucket locks. When both per-bucket lock and
 * connection lock need to be held, the per-bucket lock should be acquired
 * first, followed by the connection lock.
 *
 * All functions doing search in one of these tables increment a reference
 * counter on the connection found (if any). This reference should be dropped
 * when the caller has finished processing the connection.
 *
 *
 * INTERFACES:
 * ===========
 *
 * Connection Lookup:
 * ------------------
 *
 * conn_t *ipcl_classify_v4(mp, protocol, hdr_len, ira, ip_stack)
 * conn_t *ipcl_classify_v6(mp, protocol, hdr_len, ira, ip_stack)
 *
 * Finds connection for an incoming IPv4 or IPv6 packet. Returns NULL if
 * it can't find any associated connection. If the connection is found, its
 * reference counter is incremented.
 *
 *      mp:     mblock, containing packet header. The full header should fit
 *              into a single mblock. It should also contain at least full IP
 *              and TCP or UDP header.
 *
 *      protocol: Either IPPROTO_TCP or IPPROTO_UDP.
 *
 *      hdr_len: The size of IP header. It is used to find TCP or UDP header in
 *               the packet.
 *
 *      ira->ira_zoneid: The zone in which the returned connection must be; the
 *              zoneid corresponding to the ire_zoneid on the IRE located for
 *              the packet's destination address.
 *
 *      ira->ira_flags: Contains the IRAF_TX_MAC_EXEMPTABLE and
 *              IRAF_TX_SHARED_ADDR flags
 *
 *      For TCP connections, the lookup order is as follows:
 *              5-tuple {src, dst, protocol, local port, remote port}
 *                      lookup in ipcl_conn_fanout table.
 *              3-tuple {dst, remote port, protocol} lookup in
 *                      ipcl_bind_fanout table.
 *
 *      For UDP connections, a 5-tuple {src, dst, protocol, local port,
 *      remote port} lookup is done on ipcl_udp_fanout. Note that,
 *      these interfaces do not handle cases where a packets belongs
 *      to multiple UDP clients, which is handled in IP itself.
 *
 * If the destination IRE is ALL_ZONES (indicated by zoneid), then we must
 * determine which actual zone gets the segment.  This is used only in a
 * labeled environment.  The matching rules are:
 *
 *      - If it's not a multilevel port, then the label on the packet selects
```

```
128  *        the zone.  Unlabeled packets are delivered to the global zone.
129  *
130  *      - If it's a multilevel port, then only the zone registered to receive
131  *        packets on that port matches.
132  *
133  * Also, in a labeled environment, packet labels need to be checked.  For fully
134  * bound TCP connections, we can assume that the packet label was checked
135  * during connection establishment, and doesn't need to be checked on each
136  * packet.  For others, though, we need to check for strict equality or, for
137  * multilevel ports, membership in the range or set.  This part currently does
138  * a tnrh lookup on each packet, but could be optimized to use cached results
139  * if that were necessary.  (SCTP doesn't come through here, but if it did,
140  * we would apply the same rules as TCP.)
141  *
142  * An implication of the above is that fully-bound TCP sockets must always use
143  * distinct 4-tuples; they can't be discriminated by label alone.
144  *
145  * Note that we cannot trust labels on packets sent to fully-bound UDP sockets,
146  * as there's no connection set-up handshake and no shared state.
147  *
148  * Labels on looped-back packets within a single zone do not need to be
149  * checked, as all processes in the same zone have the same label.
150  *
151  * Finally, for unlabeled packets received by a labeled system, special rules
152  * apply.  We consider only the MLP if there is one.  Otherwise, we prefer a
153  * socket in the zone whose label matches the default label of the sender, if
154  * any.  In any event, the receiving socket must have SO_MAC_EXEMPT set and the
155  * receiver's label must dominate the sender's default label.
156  *
157  * conn_t *ipcl_tcp_lookup_reversed_ipv4(ipha_t *, tcpha_t *, int, ip_stack);
158  * conn_t *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpha_t *, int, uint_t,
159  *                                       ip_stack);
160  *
161  *      Lookup routine to find a exact match for {src, dst, local port,
162  *      remote port) for TCP connections in ipcl_conn_fanout. The address and
163  *      ports are read from the IP and TCP header respectively.
164  *
165  * conn_t       *ipcl_lookup_listener_v4(lport, laddr, protocol,
166  *                                       zoneid, ip_stack);
167  * conn_t       *ipcl_lookup_listener_v6(lport, laddr, protocol, ifindex,
168  *                                       zoneid, ip_stack);
169  *
170  *      Lookup routine to find a listener with the tuple {lport, laddr,
171  *      protocol} in the ipcl_bind_fanout table. For IPv6, an additional
172  *      parameter interface index is also compared.
173  *
174  * void ipcl_walk(func, arg, ip_stack)
175  *
176  *      Apply 'func' to every connection available. The 'func' is called as
177  *      (*func)(connp, arg). The walk is non-atomic so connections may be
178  *      created and destroyed during the walk. The CONN_CONDEMNED and
179  *      CONN_INCIPIENT flags ensure that connections which are newly created
180  *      or being destroyed are not selected by the walker.
181  *
182  * Table Updates
183  * -------------
184  *
185  * int ipcl_conn_insert(connp);
186  * int ipcl_conn_insert_v4(connp);
187  * int ipcl_conn_insert_v6(connp);
188  *
189  *      Insert 'connp' in the ipcl_conn_fanout.
190  *      Arguements :
191  *              connp           conn_t to be inserted
192  *
193  *      Return value :
```



```
194  *              0               if connp was inserted
195  *              EADDRINUSE      if the connection with the same tuple
196  *                              already exists.
197  *
198  * int ipcl_bind_insert(connp);
199  * int ipcl_bind_insert_v4(connp);
200  * int ipcl_bind_insert_v6(connp);
201  *
202  *      Insert 'connp' in ipcl_bind_fanout.
203  *      Arguements :
204  *              connp           conn_t to be inserted
205  *
206  *
207  * void ipcl_hash_remove(connp);
208  *
209  *      Removes the 'connp' from the connection fanout table.
210  *
211  * Connection Creation/Destruction
212  * -------------------------------
213  *
214  * conn_t *ipcl_conn_create(type, sleep, netstack_t *)
215  *
216  *      Creates a new conn based on the type flag, inserts it into
217  *      globalhash table.
218  *
219  *      type:   This flag determines the type of conn_t which needs to be
220  *              created i.e., which kmem_cache it comes from.
221  *              IPCL_TCPCONN    indicates a TCP connection
222  *              IPCL_SCTPCONN   indicates a SCTP connection
223  *              IPCL_UDPCONN    indicates a UDP conn_t.
224  *              IPCL_RAWIPCONN  indicates a RAWIP/ICMP conn_t.
225  *              IPCL_RTSCONN    indicates a RTS conn_t.
226  *              IPCL_DCCPCONN   indicates a DCCP conn_t.
227 #endif /* ! codereview */
228  *              IPCL_IPCCONN    indicates all other connections.
229  *
230  * void ipcl_conn_destroy(connp)
231  *
232  *      Destroys the connection state, removes it from the global
233  *      connection hash table and frees its memory.
234  */

236 #include <sys/types.h>
237 #include <sys/stream.h>
238 #include <sys/stropts.h>
239 #include <sys/sysmacros.h>
240 #include <sys/strsubr.h>
241 #include <sys/strsun.h>
242 #define _SUN_TPI_VERSION 2
243 #include <sys/ddi.h>
244 #include <sys/cmn_err.h>
245 #include <sys/debug.h>

247 #include <sys/systm.h>
248 #include <sys/param.h>
249 #include <sys/kmem.h>
250 #include <sys/isa_defs.h>
251 #include <inet/common.h>
252 #include <netinet/ip6.h>
253 #include <netinet/icmp6.h>

255 #include <inet/ip.h>
256 #include <inet/ip_if.h>
257 #include <inet/ip_ire.h>
258 #include <inet/ip6.h>
259 #include <inet/ip_ndp.h>
```

```
#include <inet/ip_impl.h>
#include <inet/udp_impl.h>
#include <inet/dccp/dccp_impl.h>
#endif /* ! codereview */
#include <inet/sctp_ip.h>
#include <inet/sctp/sctp_impl.h>
#include <inet/rawip_impl.h>
#include <inet/rts_impl.h>
#include <inet/iptun/iptun_impl.h>

#include <sys/cpuvar.h>

#include <inet/ipclassifier.h>
#include <inet/tcp.h>
#include <inet/ipsec_impl.h>

#include <sys/tsol/tnet.h>
#include <sys/sockio.h>

/* Old value for compatibility. Setable in /etc/system */
uint_t tcp_conn_hash_size = 0;

/* New value. Zero means choose automatically.  Setable in /etc/system */
uint_t ipcl_conn_hash_size = 0;
uint_t ipcl_conn_hash_memfactor = 8192;
uint_t ipcl_conn_hash_maxsize = 82500;

/* bind/dccp/udp fanout table size */
/* bind/udp fanout table size */
uint_t ipcl_bind_fanout_size = 512;
uint_t ipcl_dccp_fanout_size = 512;
#endif /* ! codereview */
uint_t ipcl_udp_fanout_size = 16384;

/* Raw socket fanout size.  Must be a power of 2. */
uint_t ipcl_raw_fanout_size = 256;

/*
 * The IPCL_IPTUN_HASH() function works best with a prime table size.  We
 * expect that most large deployments would have hundreds of tunnels, and
 * thousands in the extreme case.
 */
uint_t ipcl_iptun_fanout_size = 6143;

/*
 * Power of 2^N Primes useful for hashing for N of 0-28,
 * these primes are the nearest prime <= 2^N - 2^(N-2).
 */

#define P2Ps() {0, 0, 0, 5, 11, 23, 47, 89, 191, 383, 761, 1531, 3067,  \
                6143, 12281, 24571, 49139, 98299, 196597, 393209,       \
                786431, 1572853, 3145721, 6291449, 12582893, 25165813,  \
                50331599, 100663291, 201326557, 0}

/*
 * wrapper structure to ensure that conn and what follows it (tcp_t, etc)
 * are aligned on cache lines.
 */
typedef union itc_s {
        conn_t  itc_conn;
        char    itcu_filler[CACHE_ALIGN(conn_s)];
} itc_t;

struct kmem_cache  *tcp_conn_cache;
struct kmem_cache  *ip_conn_cache;
extern struct kmem_cache  *sctp_conn_cache;
```



```
struct kmem_cache  *udp_conn_cache;
struct kmem_cache  *rawip_conn_cache;
struct kmem_cache  *rts_conn_cache;
struct kmem_cache  *dccp_conn_cache;
#endif /* ! codereview */

extern void     tcp_timermp_free(tcp_t *);
extern mblk_t   *tcp_timermp_alloc(int);

static int      ip_conn_constructor(void *, void *, int);
static void     ip_conn_destructor(void *, void *);

static int      tcp_conn_constructor(void *, void *, int);
static void     tcp_conn_destructor(void *, void *);

static int      udp_conn_constructor(void *, void *, int);
static void     udp_conn_destructor(void *, void *);

static int      rawip_conn_constructor(void *, void *, int);
static void     rawip_conn_destructor(void *, void *);

static int      rts_conn_constructor(void *, void *, int);
static void     rts_conn_destructor(void *, void *);

static int      dccp_conn_constructor(void *, void *, int);
static void     dccp_conn_destructor(void *, void *);

#endif /* ! codereview */
/*
 * Global (for all stack instances) init routine
 */
void
ipcl_g_init(void)
{
        ip_conn_cache = kmem_cache_create("ip_conn_cache",
            sizeof (conn_t), CACHE_ALIGN_SIZE,
            ip_conn_constructor, ip_conn_destructor,
            NULL, NULL, NULL, 0);

        tcp_conn_cache = kmem_cache_create("tcp_conn_cache",
            sizeof (itc_t) + sizeof (tcp_t), CACHE_ALIGN_SIZE,
            tcp_conn_constructor, tcp_conn_destructor,
            tcp_conn_reclaim, NULL, NULL, 0);

        udp_conn_cache = kmem_cache_create("udp_conn_cache",
            sizeof (itc_t) + sizeof (udp_t), CACHE_ALIGN_SIZE,
            udp_conn_constructor, udp_conn_destructor,
            NULL, NULL, NULL, 0);

        rawip_conn_cache = kmem_cache_create("rawip_conn_cache",
            sizeof (itc_t) + sizeof (icmp_t), CACHE_ALIGN_SIZE,
            rawip_conn_constructor, rawip_conn_destructor,
            NULL, NULL, NULL, 0);

        rts_conn_cache = kmem_cache_create("rts_conn_cache",
            sizeof (itc_t) + sizeof (rts_t), CACHE_ALIGN_SIZE,
            rts_conn_constructor, rts_conn_destructor,
            NULL, NULL, NULL, 0);

        /* XXX:DCCP reclaim */
        dccp_conn_cache = kmem_cache_create("dccp_conn_cache",
            sizeof (itc_t) + sizeof (dccp_t), CACHE_ALIGN_SIZE,
            dccp_conn_constructor, dccp_conn_destructor,
            NULL, NULL, NULL, 0);
#endif /* ! codereview */
}
```

```
392 /*
393  * ipclassifier intialization routine, sets up hash tables.
394  */
395 void
396 ipcl_init(ip_stack_t *ipst)
397 {
398         int i;
399         int sizes[] = P2Ps();

401         /*
402          * Calculate size of conn fanout table from /etc/system settings
403          */
404         if (ipcl_conn_hash_size != 0) {
405                 ipst->ips_ipcl_conn_fanout_size = ipcl_conn_hash_size;
406         } else if (tcp_conn_hash_size != 0) {
407                 ipst->ips_ipcl_conn_fanout_size = tcp_conn_hash_size;
408         } else {
409                 extern pgcnt_t freemem;

411                 ipst->ips_ipcl_conn_fanout_size =
412                     (freemem * PAGESIZE) / ipcl_conn_hash_memfactor;

414                 if (ipst->ips_ipcl_conn_fanout_size > ipcl_conn_hash_maxsize) {
415                         ipst->ips_ipcl_conn_fanout_size =
416                             ipcl_conn_hash_maxsize;
417                 }
418         }

420         for (i = 9; i < sizeof (sizes) / sizeof (*sizes) - 1; i++) {
421                 if (sizes[i] >= ipst->ips_ipcl_conn_fanout_size) {
422                         break;
423                 }
424         }
425         if ((ipst->ips_ipcl_conn_fanout_size = sizes[i]) == 0) {
426                 /* Out of range, use the 2^16 value */
427                 ipst->ips_ipcl_conn_fanout_size = sizes[16];
428         }

430         /* Take values from /etc/system */
431         ipst->ips_ipcl_bind_fanout_size = ipcl_bind_fanout_size;
432         ipst->ips_ipcl_dccp_fanout_size = ipcl_dccp_fanout_size;
433 #endif /* ! codereview */
434         ipst->ips_ipcl_udp_fanout_size = ipcl_udp_fanout_size;
435         ipst->ips_ipcl_raw_fanout_size = ipcl_raw_fanout_size;
436         ipst->ips_ipcl_iptun_fanout_size = ipcl_iptun_fanout_size;

438         ASSERT(ipst->ips_ipcl_conn_fanout == NULL);

440         ipst->ips_ipcl_conn_fanout = kmem_zalloc(
441             ipst->ips_ipcl_conn_fanout_size * sizeof (connf_t), KM_SLEEP);

443         for (i = 0; i < ipst->ips_ipcl_conn_fanout_size; i++) {
444                 mutex_init(&ipst->ips_ipcl_conn_fanout[i].connf_lock, NULL,
445                     MUTEX_DEFAULT, NULL);
446         }

448         ipst->ips_ipcl_bind_fanout = kmem_zalloc(
449             ipst->ips_ipcl_bind_fanout_size * sizeof (connf_t), KM_SLEEP);

451         for (i = 0; i < ipst->ips_ipcl_bind_fanout_size; i++) {
452                 mutex_init(&ipst->ips_ipcl_bind_fanout[i].connf_lock, NULL,
453                     MUTEX_DEFAULT, NULL);
454         }

456         ipst->ips_ipcl_proto_fanout_v4 = kmem_zalloc(IPPROTO_MAX *
```



```
457             sizeof (connf_t), KM_SLEEP);
458         for (i = 0; i < IPPROTO_MAX; i++) {
459                 mutex_init(&ipst->ips_ipcl_proto_fanout_v4[i].connf_lock, NULL,
460                     MUTEX_DEFAULT, NULL);
461         }

463         ipst->ips_ipcl_proto_fanout_v6 = kmem_zalloc(IPPROTO_MAX *
464             sizeof (connf_t), KM_SLEEP);
465         for (i = 0; i < IPPROTO_MAX; i++) {
466                 mutex_init(&ipst->ips_ipcl_proto_fanout_v6[i].connf_lock, NULL,
467                     MUTEX_DEFAULT, NULL);
468         }

470         ipst->ips_rts_clients = kmem_zalloc(sizeof (connf_t), KM_SLEEP);
471         mutex_init(&ipst->ips_rts_clients->connf_lock,
472             NULL, MUTEX_DEFAULT, NULL);

474         ipst->ips_ipcl_udp_fanout = kmem_zalloc(
475             ipst->ips_ipcl_udp_fanout_size * sizeof (connf_t), KM_SLEEP);
476         for (i = 0; i < ipst->ips_ipcl_udp_fanout_size; i++) {
477                 mutex_init(&ipst->ips_ipcl_udp_fanout[i].connf_lock, NULL,
478                     MUTEX_DEFAULT, NULL);
479         }

481         ipst->ips_ipcl_iptun_fanout = kmem_zalloc(
482             ipst->ips_ipcl_iptun_fanout_size * sizeof (connf_t), KM_SLEEP);
483         for (i = 0; i < ipst->ips_ipcl_iptun_fanout_size; i++) {
484                 mutex_init(&ipst->ips_ipcl_iptun_fanout[i].connf_lock, NULL,
485                     MUTEX_DEFAULT, NULL);
486         }

488         ipst->ips_ipcl_raw_fanout = kmem_zalloc(
489             ipst->ips_ipcl_raw_fanout_size * sizeof (connf_t), KM_SLEEP);
490         for (i = 0; i < ipst->ips_ipcl_raw_fanout_size; i++) {
491                 mutex_init(&ipst->ips_ipcl_raw_fanout[i].connf_lock, NULL,
492                     MUTEX_DEFAULT, NULL);
493         }

495         ipst->ips_ipcl_globalhash_fanout = kmem_zalloc(
496             sizeof (connf_t) * CONN_G_HASH_SIZE, KM_SLEEP);
497         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
498                 mutex_init(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock,
499                     NULL, MUTEX_DEFAULT, NULL);
500         }

502         ipst->ips_ipcl_dccp_fanout = kmem_zalloc(
503             ipst->ips_ipcl_dccp_fanout_size * sizeof (connf_t), KM_SLEEP);
504         for (i = 0; i < ipst->ips_ipcl_dccp_fanout_size; i++) {
505                 mutex_init(&ipst->ips_ipcl_dccp_fanout[i].connf_lock, NULL,
506                     MUTEX_DEFAULT, NULL);
507         }
508 #endif /* ! codereview */
509 }

511 void
512 ipcl_g_destroy(void)
513 {
514         kmem_cache_destroy(ip_conn_cache);
515         kmem_cache_destroy(tcp_conn_cache);
516         kmem_cache_destroy(udp_conn_cache);
517         kmem_cache_destroy(rawip_conn_cache);
518         kmem_cache_destroy(rts_conn_cache);
519         kmem_cache_destroy(dccp_conn_cache);
520 #endif /* ! codereview */
521 }
```

```
/*
 * All user-level and kernel use of the stack must be gone
 * by now.
 */
void
ipcl_destroy(ip_stack_t *ipst)
{
        int i;

        for (i = 0; i < ipst->ips_ipcl_conn_fanout_size; i++) {
                ASSERT(ipst->ips_ipcl_conn_fanout[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_conn_fanout[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_conn_fanout, ipst->ips_ipcl_conn_fanout_size *
            sizeof (connf_t));
        ipst->ips_ipcl_conn_fanout = NULL;

        for (i = 0; i < ipst->ips_ipcl_bind_fanout_size; i++) {
                ASSERT(ipst->ips_ipcl_bind_fanout[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_bind_fanout[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_bind_fanout, ipst->ips_ipcl_bind_fanout_size *
            sizeof (connf_t));
        ipst->ips_ipcl_bind_fanout = NULL;

        for (i = 0; i < IPPROTO_MAX; i++) {
                ASSERT(ipst->ips_ipcl_proto_fanout_v4[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_proto_fanout_v4[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_proto_fanout_v4,
            IPPROTO_MAX * sizeof (connf_t));
        ipst->ips_ipcl_proto_fanout_v4 = NULL;

        for (i = 0; i < IPPROTO_MAX; i++) {
                ASSERT(ipst->ips_ipcl_proto_fanout_v6[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_proto_fanout_v6[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_proto_fanout_v6,
            IPPROTO_MAX * sizeof (connf_t));
        ipst->ips_ipcl_proto_fanout_v6 = NULL;

        for (i = 0; i < ipst->ips_ipcl_udp_fanout_size; i++) {
                ASSERT(ipst->ips_ipcl_udp_fanout[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_udp_fanout[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_udp_fanout, ipst->ips_ipcl_udp_fanout_size *
            sizeof (connf_t));
        ipst->ips_ipcl_udp_fanout = NULL;

        for (i = 0; i < ipst->ips_ipcl_iptun_fanout_size; i++) {
                ASSERT(ipst->ips_ipcl_iptun_fanout[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_iptun_fanout[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_iptun_fanout,
            ipst->ips_ipcl_iptun_fanout_size * sizeof (connf_t));
        ipst->ips_ipcl_iptun_fanout = NULL;

        for (i = 0; i < ipst->ips_ipcl_raw_fanout_size; i++) {
                ASSERT(ipst->ips_ipcl_raw_fanout[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_raw_fanout[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_raw_fanout, ipst->ips_ipcl_raw_fanout_size *
            sizeof (connf_t));
        ipst->ips_ipcl_raw_fanout = NULL;

        for (i = 0; i < CONN_G_HASH_SIZE; i++) {
```



```
                ASSERT(ipst->ips_ipcl_globalhash_fanout[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_globalhash_fanout,
            sizeof (connf_t) * CONN_G_HASH_SIZE);
        ipst->ips_ipcl_globalhash_fanout = NULL;

        for (i = 0; i < ipst->ips_ipcl_dccp_fanout_size; i++) {
                ASSERT(ipst->ips_ipcl_dccp_fanout[i].connf_head == NULL);
                mutex_destroy(&ipst->ips_ipcl_dccp_fanout[i].connf_lock);
        }
        kmem_free(ipst->ips_ipcl_dccp_fanout, ipst->ips_ipcl_dccp_fanout_size *
            sizeof (connf_t));
        ipst->ips_ipcl_dccp_fanout = NULL;

#endif /* ! codereview */
        ASSERT(ipst->ips_rts_clients->connf_head == NULL);
        mutex_destroy(&ipst->ips_rts_clients->connf_lock);
        kmem_free(ipst->ips_rts_clients, sizeof (connf_t));
        ipst->ips_rts_clients = NULL;
}

/*
 * conn creation routine. initialize the conn, sets the reference
 * and inserts it in the global hash table.
 */
conn_t *
ipcl_conn_create(uint32_t type, int sleep, netstack_t *ns)
{
        conn_t  *connp;
        struct kmem_cache *conn_cache;

        switch (type) {
        case IPCL_SCTPCONN:
                if ((connp = kmem_cache_alloc(sctp_conn_cache, sleep)) == NULL)
                        return (NULL);
                sctp_conn_init(connp);
                netstack_hold(ns);
                connp->conn_netstack = ns;
                connp->conn_ixa->ixa_ipst = ns->netstack_ip;
                connp->conn_ixa->ixa_conn_id = (long)connp;
                ipcl_globalhash_insert(connp);
                return (connp);

        case IPCL_TCPCONN:
                conn_cache = tcp_conn_cache;
                break;

        case IPCL_UDPCONN:
                conn_cache = udp_conn_cache;
                break;

        case IPCL_RAWIPCONN:
                conn_cache = rawip_conn_cache;
                break;

        case IPCL_RTSCONN:
                conn_cache = rts_conn_cache;
                break;

        case IPCL_IPCCONN:
                conn_cache = ip_conn_cache;
                break;

        case IPCL_DCCPCONN:
                conn_cache = dccp_conn_cache;
```

```
655                 break;

657 #endif /* ! codereview */
658         default:
659                 connp = NULL;
660                 ASSERT(0);
661         }

663         if ((connp = kmem_cache_alloc(conn_cache, sleep)) == NULL)
664                 return (NULL);

666         connp->conn_ref = 1;
667         netstack_hold(ns);
668         connp->conn_netstack = ns;
669         connp->conn_ixa->ixa_ipst = ns->netstack_ip;
670         connp->conn_ixa->ixa_conn_id = (long)connp;
671         ipcl_globalhash_insert(connp);
672         return (connp);
673 }

675 void
676 ipcl_conn_destroy(conn_t *connp)
677 {
678         mblk_t  *mp;
679         netstack_t      *ns = connp->conn_netstack;

681         ASSERT(!MUTEX_HELD(&connp->conn_lock));
682         ASSERT(connp->conn_ref == 0);
683         ASSERT(connp->conn_ioctlref == 0);

685         DTRACE_PROBE1(conn__destroy, conn_t *, connp);

687         if (connp->conn_cred != NULL) {
688                 crfree(connp->conn_cred);
689                 connp->conn_cred = NULL;
690                 /* ixa_cred done in ipcl_conn_cleanup below */
691         }

693         if (connp->conn_ht_iphc != NULL) {
694                 kmem_free(connp->conn_ht_iphc, connp->conn_ht_iphc_allocated);
695                 connp->conn_ht_iphc = NULL;
696                 connp->conn_ht_iphc_allocated = 0;
697                 connp->conn_ht_iphc_len = 0;
698                 connp->conn_ht_ulp = NULL;
699                 connp->conn_ht_ulp_len = 0;
700         }
701         ip_pkt_free(&connp->conn_xmit_ipp);

703         ipcl_globalhash_remove(connp);

705         if (connp->conn_latch != NULL) {
706                 IPLATCH_REFRELE(connp->conn_latch);
707                 connp->conn_latch = NULL;
708         }
709         if (connp->conn_latch_in_policy != NULL) {
710                 IPPOL_REFRELE(connp->conn_latch_in_policy);
711                 connp->conn_latch_in_policy = NULL;
712         }
713         if (connp->conn_latch_in_action != NULL) {
714                 IPACT_REFRELE(connp->conn_latch_in_action);
715                 connp->conn_latch_in_action = NULL;
716         }
717         if (connp->conn_policy != NULL) {
718                 IPPH_REFRELE(connp->conn_policy, ns);
719                 connp->conn_policy = NULL;
720         }
```



```
722         if (connp->conn_ipsec_opt_mp != NULL) {
723                 freemsg(connp->conn_ipsec_opt_mp);
724                 connp->conn_ipsec_opt_mp = NULL;
725         }

727         if (connp->conn_flags & IPCL_TCPCONN) {
728                 tcp_t *tcp = connp->conn_tcp;

730                 tcp_free(tcp);
731                 mp = tcp->tcp_timercache;

733                 tcp->tcp_tcps = NULL;

735                 /*
736                  * tcp_rsrv_mp can be NULL if tcp_get_conn() fails to allocate
737                  * the mblk.
738                  */
739                 if (tcp->tcp_rsrv_mp != NULL) {
740                         freeb(tcp->tcp_rsrv_mp);
741                         tcp->tcp_rsrv_mp = NULL;
742                         mutex_destroy(&tcp->tcp_rsrv_mp_lock);
743                 }

745                 ipcl_conn_cleanup(connp);
746                 connp->conn_flags = IPCL_TCPCONN;
747                 if (ns != NULL) {
748                         ASSERT(tcp->tcp_tcps == NULL);
749                         connp->conn_netstack = NULL;
750                         connp->conn_ixa->ixa_ipst = NULL;
751                         netstack_rele(ns);
752                 }

754                 bzero(tcp, sizeof (tcp_t));

756                 tcp->tcp_timercache = mp;
757                 tcp->tcp_connp = connp;
758                 kmem_cache_free(tcp_conn_cache, connp);
759                 return;
760         }

762         if (connp->conn_flags & IPCL_SCTPCONN) {
763                 ASSERT(ns != NULL);
764                 sctp_free(connp);
765                 return;
766         }

768         if (connp->conn_flags & IPCL_DCCPCONN) {
769                 dccp_t  *dccp = connp->conn_dccp;

771                 cmn_err(CE_NOTE, "ipclassifier: conn_flags DCCP cache_free");

773                 /* XXX:DCCP */
774                 /* Crash bug here: udp_conn_cache and dccp_conn_cache */
775 /*
776                 ipcl_conn_cleanup(connp);
777                 connp->conn_flags = IPCL_DCCPCONN;
778                 bzero(dccp, sizeof (dccp_t));
779                 dccp->dccp_connp = connp;
780                 kmem_cache_free(dccp_conn_cache, connp);
781                 return;
782 */
783         }

785 #endif /* ! codereview */
786         ipcl_conn_cleanup(connp);
```

```
787         if (ns != NULL) {
788                 connp->conn_netstack = NULL;
789                 connp->conn_ixa->ixa_ipst = NULL;
790                 netstack_rele(ns);
791         }

793         /* leave conn_priv aka conn_udp, conn_icmp, etc in place. */
794         if (connp->conn_flags & IPCL_UDPCONN) {
795                 connp->conn_flags = IPCL_UDPCONN;
796                 kmem_cache_free(udp_conn_cache, connp);
797         } else if (connp->conn_flags & IPCL_RAWIPCONN) {
798                 connp->conn_flags = IPCL_RAWIPCONN;
799                 connp->conn_proto = IPPROTO_ICMP;
800                 connp->conn_ixa->ixa_protocol = connp->conn_proto;
801                 kmem_cache_free(rawip_conn_cache, connp);
802         } else if (connp->conn_flags & IPCL_RTSCONN) {
803                 connp->conn_flags = IPCL_RTSCONN;
804                 kmem_cache_free(rts_conn_cache, connp);
805         } else {
806                 connp->conn_flags = IPCL_IPCCONN;
807                 ASSERT(connp->conn_flags & IPCL_IPCCONN);
808                 ASSERT(connp->conn_priv == NULL);
809                 kmem_cache_free(ip_conn_cache, connp);
810         }
811 }

813 /*
814  * Running in cluster mode - deregister listener information
815  */
816 static void
817 ipcl_conn_unlisten(conn_t *connp)
818 {
819         ASSERT((connp->conn_flags & IPCL_CL_LISTENER) != 0);
820         ASSERT(connp->conn_lport != 0);

822         if (cl_inet_unlisten != NULL) {
823                 sa_family_t     addr_family;
824                 uint8_t         *laddrp;

826                 if (connp->conn_ipversion == IPV6_VERSION) {
827                         addr_family = AF_INET6;
828                         laddrp = (uint8_t *)&connp->conn_bound_addr_v6;
829                 } else {
830                         addr_family = AF_INET;
831                         laddrp = (uint8_t *)&connp->conn_bound_addr_v4;
832                 }
833                 (*cl_inet_unlisten)(connp->conn_netstack->netstack_stackid,
834                     IPPROTO_TCP, addr_family, laddrp, connp->conn_lport, NULL);
835         }
836         connp->conn_flags &= ~IPCL_CL_LISTENER;
837 }

839 /*
840  * We set the IPCL_REMOVED flag (instead of clearing the flag indicating
841  * which table the conn belonged to). So for debugging we can see which hash
842  * table this connection was in.
843  */
844 #define IPCL_HASH_REMOVE(connp) {                                       \
845         connf_t *connfp = (connp)->conn_fanout;                         \
846         ASSERT(!MUTEX_HELD(&((connp)->conn_lock)));                     \
847         if (connfp != NULL) {                                           \
848                 mutex_enter(&connfp->connf_lock);                       \
849                 if ((connp)->conn_next != NULL)                         \
850                         (connp)->conn_next->conn_prev =                 \
851                             (connp)->conn_prev;                         \
852                 if ((connp)->conn_prev != NULL)                         \
```



```
853                         (connp)->conn_prev->conn_next =                 \
854                             (connp)->conn_next;                         \
855                 else                                                    \
856                         connfp->connf_head = (connp)->conn_next;        \
857                 (connp)->conn_fanout = NULL;                            \
858                 (connp)->conn_next = NULL;                              \
859                 (connp)->conn_prev = NULL;                              \
860                 (connp)->conn_flags |= IPCL_REMOVED;                    \
861                 if (((connp)->conn_flags & IPCL_CL_LISTENER) != 0)      \
862                         ipcl_conn_unlisten((connp));                    \
863                 CONN_DEC_REF((connp));                                  \
864                 mutex_exit(&connfp->connf_lock);                        \
865         }                                                               \
866 }

868 void
869 ipcl_hash_remove(conn_t *connp)
870 {
871         uint8_t         protocol = connp->conn_proto;

873         IPCL_HASH_REMOVE(connp);
874         if (protocol == IPPROTO_RSVP)
875                 ill_set_inputfn_all(connp->conn_netstack->netstack_ip);
876 }

878 /*
879  * The whole purpose of this function is allow removal of
880  * a conn_t from the connected hash for timewait reclaim.
881  * This is essentially a TW reclaim fastpath where timewait
882  * collector checks under fanout lock (so no one else can
883  * get access to the conn_t) that refcnt is 2 i.e. one for
884  * TCP and one for the classifier hash list. If ref count
885  * is indeed 2, we can just remove the conn under lock and
886  * avoid cleaning up the conn under squeue. This gives us
887  * improved performance.
888  */
889 void
890 ipcl_hash_remove_locked(conn_t *connp, connf_t  *connfp)
891 {
892         ASSERT(MUTEX_HELD(&connfp->connf_lock));
893         ASSERT(MUTEX_HELD(&connp->conn_lock));
894         ASSERT((connp->conn_flags & IPCL_CL_LISTENER) == 0);

896         if ((connp)->conn_next != NULL) {
897                 (connp)->conn_next->conn_prev = (connp)->conn_prev;
898         }
899         if ((connp)->conn_prev != NULL) {
900                 (connp)->conn_prev->conn_next = (connp)->conn_next;
901         } else {
902                 connfp->connf_head = (connp)->conn_next;
903         }
904         (connp)->conn_fanout = NULL;
905         (connp)->conn_next = NULL;
906         (connp)->conn_prev = NULL;
907         (connp)->conn_flags |= IPCL_REMOVED;
908         ASSERT((connp)->conn_ref == 2);
909         (connp)->conn_ref--;
910 }

912 #define IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp) {              \
913         ASSERT((connp)->conn_fanout == NULL);                           \
914         ASSERT((connp)->conn_next == NULL);                             \
915         ASSERT((connp)->conn_prev == NULL);                             \
916         if ((connfp)->connf_head != NULL) {                             \
917                 (connfp)->connf_head->conn_prev = (connp);              \
918                 (connp)->conn_next = (connfp)->connf_head;              \
```

```
919         }                                                               \
920         (connp)->conn_fanout = (connfp);                                \
921         (connfp)->connf_head = (connp);                                 \
922         (connp)->conn_flags = ((connp)->conn_flags & ~IPCL_REMOVED) |    \
923             IPCL_CONNECTED;                                             \
924         CONN_INC_REF(connp);                                            \
925 }

927 #define IPCL_HASH_INSERT_CONNECTED(connfp, connp) {                     \
928         IPCL_HASH_REMOVE((connp));                                      \
929         mutex_enter(&(connfp)->connf_lock);                             \
930         IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);               \
931         mutex_exit(&(connfp)->connf_lock);                              \
932 }

934 #define IPCL_HASH_INSERT_BOUND(connfp, connp) {                         \
935         conn_t *pconnp = NULL, *nconnp;                                 \
936         IPCL_HASH_REMOVE((connp));                                      \
937         mutex_enter(&(connfp)->connf_lock);                             \
938         nconnp = (connfp)->connf_head;                                  \
939         while (nconnp != NULL &&                                        \
940             !_IPCL_V4_MATCH_ANY(nconnp->conn_laddr_v6)) {               \
941                 pconnp = nconnp;                                        \
942                 nconnp = nconnp->conn_next;                             \
943         }                                                               \
944         if (pconnp != NULL) {                                           \
945                 pconnp->conn_next = (connp);                            \
946                 (connp)->conn_prev = pconnp;                            \
947         } else {                                                        \
948                 (connfp)->connf_head = (connp);                         \
949         }                                                               \
950         if (nconnp != NULL) {                                           \
951                 (connp)->conn_next = nconnp;                            \
952                 nconnp->conn_prev = (connp);                            \
953         }                                                               \
954         (connp)->conn_fanout = (connfp);                                \
955         (connp)->conn_flags = ((connp)->conn_flags & ~IPCL_REMOVED) |    \
956             IPCL_BOUND;                                                 \
957         CONN_INC_REF(connp);                                            \
958         mutex_exit(&(connfp)->connf_lock);                              \
959 }

961 #define IPCL_HASH_INSERT_WILDCARD(connfp, connp) {                      \
962         conn_t **list, *prev, *next;                                    \
963         boolean_t isv4mapped =                                          \
964             IN6_IS_ADDR_V4MAPPED(&(connp)->conn_laddr_v6);              \
965         IPCL_HASH_REMOVE((connp));                                      \
966         mutex_enter(&(connfp)->connf_lock);                             \
967         list = &(connfp)->connf_head;                                   \
968         prev = NULL;                                                    \
969         while ((next = *list) != NULL) {                                \
970                 if (isv4mapped &&                                       \
971                     IN6_IS_ADDR_UNSPECIFIED(&next->conn_laddr_v6) &&    \
972                     connp->conn_zoneid == next->conn_zoneid) {          \
973                         (connp)->conn_next = next;                      \
974                         if (prev != NULL)                               \
975                                 prev = next->conn_prev;                 \
976                         next->conn_prev = (connp);                      \
977                         break;                                          \
978                 }                                                       \
979                 list = &next->conn_next;                                \
980                 prev = next;                                            \
981         }                                                               \
982         (connp)->conn_prev = prev;                                      \
983         *list = (connp);                                                \
984         (connp)->conn_fanout = (connfp);                                \
```



```
985         (connp)->conn_flags = ((connp)->conn_flags & ~IPCL_REMOVED) |   \
986             IPCL_BOUND;                                                 \
987         CONN_INC_REF((connp));                                          \
988         mutex_exit(&(connfp)->connf_lock);                              \
989 }

991 void
992 ipcl_hash_insert_wildcard(connf_t *connfp, conn_t *connp)
993 {
994         IPCL_HASH_INSERT_WILDCARD(connfp, connp);
995 }

997 /*
998  * Because the classifier is used to classify inbound packets, the destination
999  * address is meant to be our local tunnel address (tunnel source), and the
1000 * source the remote tunnel address (tunnel destination).
1001 *
1002 * Note that conn_proto can't be used for fanout since the upper protocol
1003 * can be both 41 and 4 when IPv6 and IPv4 are over the same tunnel.
1004 */
1005 conn_t *
1006 ipcl_iptun_classify_v4(ipaddr_t *src, ipaddr_t *dst, ip_stack_t *ipst)
1007 {
1008         connf_t *connfp;
1009         conn_t  *connp;

1011         /* first look for IPv4 tunnel links */
1012         connfp = &ipst->ips_ipcl_iptun_fanout[IPCL_IPTUN_HASH(*dst, *src)];
1013         mutex_enter(&connfp->connf_lock);
1014         for (connp = connfp->connf_head; connp != NULL;
1015             connp = connp->conn_next) {
1016                 if (IPCL_IPTUN_MATCH(connp, *dst, *src))
1017                         break;
1018         }
1019         if (connp != NULL)
1020                 goto done;

1022         mutex_exit(&connfp->connf_lock);

1024         /* We didn't find an IPv4 tunnel, try a 6to4 tunnel */
1025         connfp = &ipst->ips_ipcl_iptun_fanout[IPCL_IPTUN_HASH(*dst,
1026             INADDR_ANY)];
1027         mutex_enter(&connfp->connf_lock);
1028         for (connp = connfp->connf_head; connp != NULL;
1029             connp = connp->conn_next) {
1030                 if (IPCL_IPTUN_MATCH(connp, *dst, INADDR_ANY))
1031                         break;
1032         }
1033 done:
1034         if (connp != NULL)
1035                 CONN_INC_REF(connp);
1036         mutex_exit(&connfp->connf_lock);
1037         return (connp);
1038 }

1040 conn_t *
1041 ipcl_iptun_classify_v6(in6_addr_t *src, in6_addr_t *dst, ip_stack_t *ipst)
1042 {
1043         connf_t *connfp;
1044         conn_t  *connp;

1046         /* Look for an IPv6 tunnel link */
1047         connfp = &ipst->ips_ipcl_iptun_fanout[IPCL_IPTUN_HASH_V6(dst, src)];
1048         mutex_enter(&connfp->connf_lock);
1049         for (connp = connfp->connf_head; connp != NULL;
1050             connp = connp->conn_next) {
```

```
1051                 if (IPCL_IPTUN_MATCH_V6(connp, dst, src)) {
1052                         CONN_INC_REF(connp);
1053                         break;
1054                 }
1055         }
1056         mutex_exit(&connfp->connf_lock);
1057         return (connp);
1058 }

1060 /*
1061  * This function is used only for inserting SCTP raw socket now.
1062  * This may change later.
1063  *
1064  * Note that only one raw socket can be bound to a port.  The param
1065  * lport is in network byte order.
1066  */
1067 static int
1068 ipcl_sctp_hash_insert(conn_t *connp, in_port_t lport)
1069 {
1070         connf_t *connfp;
1071         conn_t  *oconnp;
1072         ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;

1074         connfp = &ipst->ips_ipcl_raw_fanout[IPCL_RAW_HASH(ntohs(lport), ipst)];

1076         /* Check for existing raw socket already bound to the port. */
1077         mutex_enter(&connfp->connf_lock);
1078         for (oconnp = connfp->connf_head; oconnp != NULL;
1079             oconnp = oconnp->conn_next) {
1080                 if (oconnp->conn_lport == lport &&
1081                     oconnp->conn_zoneid == connp->conn_zoneid &&
1082                     oconnp->conn_family == connp->conn_family &&
1083                     ((IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6) ||
1084                     IN6_IS_ADDR_UNSPECIFIED(&oconnp->conn_laddr_v6) ||
1085                     IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_laddr_v6) ||
1086                     IN6_IS_ADDR_V4MAPPED_ANY(&oconnp->conn_laddr_v6)) ||
1087                     IN6_ARE_ADDR_EQUAL(&oconnp->conn_laddr_v6,
1088                     &connp->conn_laddr_v6))) {
1089                         break;
1090                 }
1091         }
1092         mutex_exit(&connfp->connf_lock);
1093         if (oconnp != NULL)
1094                 return (EADDRNOTAVAIL);

1096         if (IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) ||
1097             IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
1098                 if (IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6) ||
1099                     IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_laddr_v6)) {
1100                         IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1101                 } else {
1102                         IPCL_HASH_INSERT_BOUND(connfp, connp);
1103                 }
1104         } else {
1105                 IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1106         }
1107         return (0);
1108 }

1110 static int
1111 ipcl_iptun_hash_insert(conn_t *connp, ip_stack_t *ipst)
1112 {
1113         connf_t *connfp;
1114         conn_t  *tconnp;
1115         ipaddr_t laddr = connp->conn_laddr_v4;
1116         ipaddr_t faddr = connp->conn_faddr_v4;
```



```
1118         connfp = &ipst->ips_ipcl_iptun_fanout[IPCL_IPTUN_HASH(laddr, faddr)];
1119         mutex_enter(&connfp->connf_lock);
1120         for (tconnp = connfp->connf_head; tconnp != NULL;
1121             tconnp = tconnp->conn_next) {
1122                 if (IPCL_IPTUN_MATCH(tconnp, laddr, faddr)) {
1123                         /* A tunnel is already bound to these addresses. */
1124                         mutex_exit(&connfp->connf_lock);
1125                         return (EADDRINUSE);
1126                 }
1127         }
1128         IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1129         mutex_exit(&connfp->connf_lock);
1130         return (0);
1131 }

1133 static int
1134 ipcl_iptun_hash_insert_v6(conn_t *connp, ip_stack_t *ipst)
1135 {
1136         connf_t *connfp;
1137         conn_t  *tconnp;
1138         in6_addr_t *laddr = &connp->conn_laddr_v6;
1139         in6_addr_t *faddr = &connp->conn_faddr_v6;

1141         connfp = &ipst->ips_ipcl_iptun_fanout[IPCL_IPTUN_HASH_V6(laddr, faddr)];
1142         mutex_enter(&connfp->connf_lock);
1143         for (tconnp = connfp->connf_head; tconnp != NULL;
1144             tconnp = tconnp->conn_next) {
1145                 if (IPCL_IPTUN_MATCH_V6(tconnp, laddr, faddr)) {
1146                         /* A tunnel is already bound to these addresses. */
1147                         mutex_exit(&connfp->connf_lock);
1148                         return (EADDRINUSE);
1149                 }
1150         }
1151         IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1152         mutex_exit(&connfp->connf_lock);
1153         return (0);
1154 }

1156 /*
1157  * Check for a MAC exemption conflict on a labeled system.  Note that for
1158  * protocols that use port numbers (UDP, TCP, SCTP), we do this check up in the
1159  * transport layer.  This check is for binding all other protocols.
1160  *
1161  * Returns true if there's a conflict.
1162  */
1163 static boolean_t
1164 check_exempt_conflict_v4(conn_t *connp, ip_stack_t *ipst)
1165 {
1166         connf_t *connfp;
1167         conn_t *tconn;

1169         connfp = &ipst->ips_ipcl_proto_fanout_v4[connp->conn_proto];
1170         mutex_enter(&connfp->connf_lock);
1171         for (tconn = connfp->connf_head; tconn != NULL;
1172             tconn = tconn->conn_next) {
1173                 /* We don't allow v4 fallback for v6 raw socket */
1174                 if (connp->conn_family != tconn->conn_family)
1175                         continue;
1176                 /* If neither is exempt, then there's no conflict */
1177                 if ((connp->conn_mac_mode == CONN_MAC_DEFAULT) &&
1178                     (tconn->conn_mac_mode == CONN_MAC_DEFAULT))
1179                         continue;
1180                 /* We are only concerned about sockets for a different zone */
1181                 if (connp->conn_zoneid == tconn->conn_zoneid)
1182                         continue;
```

```
1183                 /* If both are bound to different specific addrs, ok */
1184                 if (connp->conn_laddr_v4 != INADDR_ANY &&
1185                     tconn->conn_laddr_v4 != INADDR_ANY &&
1186                     connp->conn_laddr_v4 != tconn->conn_laddr_v4)
1187                         continue;
1188                 /* These two conflict; fail */
1189                 break;
1190         }
1191         mutex_exit(&connfp->connf_lock);
1192         return (tconn != NULL);
1193 }

1195 static boolean_t
1196 check_exempt_conflict_v6(conn_t *connp, ip_stack_t *ipst)
1197 {
1198         connf_t *connfp;
1199         conn_t *tconn;

1201         connfp = &ipst->ips_ipcl_proto_fanout_v6[connp->conn_proto];
1202         mutex_enter(&connfp->connf_lock);
1203         for (tconn = connfp->connf_head; tconn != NULL;
1204             tconn = tconn->conn_next) {
1205                 /* We don't allow v4 fallback for v6 raw socket */
1206                 if (connp->conn_family != tconn->conn_family)
1207                         continue;
1208                 /* If neither is exempt, then there's no conflict */
1209                 if ((connp->conn_mac_mode == CONN_MAC_DEFAULT) &&
1210                     (tconn->conn_mac_mode == CONN_MAC_DEFAULT))
1211                         continue;
1212                 /* We are only concerned about sockets for a different zone */
1213                 if (connp->conn_zoneid == tconn->conn_zoneid)
1214                         continue;
1215                 /* If both are bound to different addrs, ok */
1216                 if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6) &&
1217                     !IN6_IS_ADDR_UNSPECIFIED(&tconn->conn_laddr_v6) &&
1218                     !IN6_ARE_ADDR_EQUAL(&connp->conn_laddr_v6,
1219                     &tconn->conn_laddr_v6))
1220                         continue;
1221                 /* These two conflict; fail */
1222                 break;
1223         }
1224         mutex_exit(&connfp->connf_lock);
1225         return (tconn != NULL);
1226 }

1228 /*
1229  * (v4, v6) bind hash insertion routines
1230  * The caller has already setup the conn (conn_proto, conn_laddr_v6, conn_lport)
1231  */

1233 int
1234 ipcl_bind_insert(conn_t *connp)
1235 {
1236         if (connp->conn_ipversion == IPV6_VERSION)
1237                 return (ipcl_bind_insert_v6(connp));
1238         else
1239                 return (ipcl_bind_insert_v4(connp));
1240 }

1242 int
1243 ipcl_bind_insert_v4(conn_t *connp)
1244 {
1245         connf_t *connfp;
1246         int     ret = 0;
1247         ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
1248         uint16_t        lport = connp->conn_lport;
```



```
1249         uint8_t         protocol = connp->conn_proto;

1251         if (IPCL_IS_IPTUN(connp))
1252                 return (ipcl_iptun_hash_insert(connp, ipst));

1254         switch (protocol) {
1255         default:
1256                 if (is_system_labeled() &&
1257                     check_exempt_conflict_v4(connp, ipst))
1258                         return (EADDRINUSE);
1259                 /* FALLTHROUGH */
1260         case IPPROTO_UDP:
1261                 if (protocol == IPPROTO_UDP) {
1262                         connfp = &ipst->ips_ipcl_udp_fanout[
1263                             IPCL_UDP_HASH(lport, ipst)];
1264                 } else {
1265                         connfp = &ipst->ips_ipcl_proto_fanout_v4[protocol];
1266                 }

1268                 if (connp->conn_faddr_v4 != INADDR_ANY) {
1269                         IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1270                 } else if (connp->conn_laddr_v4 != INADDR_ANY) {
1271                         IPCL_HASH_INSERT_BOUND(connfp, connp);
1272                 } else {
1273                         IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1274                 }
1275                 if (protocol == IPPROTO_RSVP)
1276                         ill_set_inputfn_all(ipst);
1277                 break;

1279         case IPPROTO_TCP:
1280                 /* Insert it in the Bind Hash */
1281                 ASSERT(connp->conn_zoneid != ALL_ZONES);
1282                 connfp = &ipst->ips_ipcl_bind_fanout[
1283                     IPCL_BIND_HASH(lport, ipst)];
1284                 if (connp->conn_laddr_v4 != INADDR_ANY) {
1285                         IPCL_HASH_INSERT_BOUND(connfp, connp);
1286                 } else {
1287                         IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1288                 }
1289                 if (cl_inet_listen != NULL) {
1290                         ASSERT(connp->conn_ipversion == IPV4_VERSION);
1291                         connp->conn_flags |= IPCL_CL_LISTENER;
1292                         (*cl_inet_listen)(
1293                             connp->conn_netstack->netstack_stackid,
1294                             IPPROTO_TCP, AF_INET,
1295                             (uint8_t *)&connp->conn_bound_addr_v4, lport, NULL);
1296                 }
1297                 break;

1299         case IPPROTO_SCTP:
1300                 ret = ipcl_sctp_hash_insert(connp, lport);
1301                 break;

1303         case IPPROTO_DCCP:
1304                 cmn_err(CE_NOTE, "ipcl_bind_insert_v4");
1305                 ASSERT(connp->conn_zoneid != ALL_ZONES);
1306                 connfp = &ipst->ips_ipcl_dccp_fanout[
1307                     IPCL_DCCP_HASH(lport, ipst)];
1308                 if (connp->conn_laddr_v4 != INADDR_ANY) {
1309                         IPCL_HASH_INSERT_BOUND(connfp, connp);
1310                 } else {
1311                         IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1312                 }
1313                 /* XXX:DCCP */
1314                 break;
```

```
#endif /* ! codereview */
        }


#endif /* ! codereview */
        return (ret);
}

int
ipcl_bind_insert_v6(conn_t *connp)
{
        connf_t         *connfp;
        int             ret = 0;
        ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
        uint16_t        lport = connp->conn_lport;
        uint8_t         protocol = connp->conn_proto;

        if (IPCL_IS_IPTUN(connp)) {
                return (ipcl_iptun_hash_insert_v6(connp, ipst));
        }

        switch (protocol) {
        default:
                if (is_system_labeled() &&
                    check_exempt_conflict_v6(connp, ipst))
                        return (EADDRINUSE);
                /* FALLTHROUGH */
        case IPPROTO_UDP:
                if (protocol == IPPROTO_UDP) {
                        connfp = &ipst->ips_ipcl_udp_fanout[
                            IPCL_UDP_HASH(lport, ipst)];
                } else {
                        connfp = &ipst->ips_ipcl_proto_fanout_v6[protocol];
                }

                if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6)) {
                        IPCL_HASH_INSERT_CONNECTED(connfp, connp);
                } else if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
                        IPCL_HASH_INSERT_BOUND(connfp, connp);
                } else {
                        IPCL_HASH_INSERT_WILDCARD(connfp, connp);
                }
                break;

        case IPPROTO_TCP:
                /* Insert it in the Bind Hash */
                ASSERT(connp->conn_zoneid != ALL_ZONES);
                connfp = &ipst->ips_ipcl_bind_fanout[
                    IPCL_BIND_HASH(lport, ipst)];
                if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
                        IPCL_HASH_INSERT_BOUND(connfp, connp);
                } else {
                        IPCL_HASH_INSERT_WILDCARD(connfp, connp);
                }
                if (cl_inet_listen != NULL) {
                        sa_family_t     addr_family;
                        uint8_t         *laddrp;

                        if (connp->conn_ipversion == IPV6_VERSION) {
                                addr_family = AF_INET6;
                                laddrp =
                                    (uint8_t *)&connp->conn_bound_addr_v6;
                        } else {
                                addr_family = AF_INET;
                                laddrp = (uint8_t *)&connp->conn_bound_addr_v4;
                        }
```



```
                        connp->conn_flags |= IPCL_CL_LISTENER;
                        (*cl_inet_listen)(
                            connp->conn_netstack->netstack_stackid,
                            IPPROTO_TCP, addr_family, laddrp, lport, NULL);
                }
                break;

        case IPPROTO_SCTP:
                ret = ipcl_sctp_hash_insert(connp, lport);
                break;

        case IPPROTO_DCCP:
                /* XXX:DCCP */
                break;
#endif /* ! codereview */
        }

        return (ret);
}

/*
 * ipcl_conn_hash insertion routines.
 * The caller has already set conn_proto and the addresses/ports in the conn_t.
 */

int
ipcl_conn_insert(conn_t *connp)
{
        if (connp->conn_ipversion == IPV6_VERSION)
                return (ipcl_conn_insert_v6(connp));
        else
                return (ipcl_conn_insert_v4(connp));
}

int
ipcl_conn_insert_v4(conn_t *connp)
{
        connf_t         *connfp;
        conn_t          *tconnp;
        int             ret = 0;
        ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
        uint16_t        lport = connp->conn_lport;
        uint8_t         protocol = connp->conn_proto;

        if (IPCL_IS_IPTUN(connp))
                return (ipcl_iptun_hash_insert(connp, ipst));

        switch (protocol) {
        case IPPROTO_TCP:
                /*
                 * For TCP, we check whether the connection tuple already
                 * exists before allowing the connection to proceed.  We
                 * also allow indexing on the zoneid. This is to allow
                 * multiple shared stack zones to have the same tcp
                 * connection tuple. In practice this only happens for
                 * INADDR_LOOPBACK as it's the only local address which
                 * doesn't have to be unique.
                 */
                connfp = &ipst->ips_ipcl_conn_fanout[
                    IPCL_CONN_HASH(connp->conn_faddr_v4,
                    connp->conn_ports, ipst)];
                mutex_enter(&connfp->connf_lock);
                for (tconnp = connfp->connf_head; tconnp != NULL;
                    tconnp = tconnp->conn_next) {
                        if (IPCL_CONN_MATCH(tconnp, connp->conn_proto,
                            connp->conn_faddr_v4, connp->conn_laddr_v4,
```

```
1447                            connp->conn_ports) &&
1448                            IPCL_ZONE_MATCH(tconnp, connp->conn_zoneid)) {
1449                                /* Already have a conn. bail out */
1450                                mutex_exit(&connfp->connf_lock);
1451                                return (EADDRINUSE);
1452                        }
1453                }
1454                if (connp->conn_fanout != NULL) {
1455                        /*
1456                         * Probably a XTI/TLI application trying to do a
1457                         * rebind. Let it happen.
1458                         */
1459                        mutex_exit(&connfp->connf_lock);
1460                        IPCL_HASH_REMOVE(connp);
1461                        mutex_enter(&connfp->connf_lock);
1462                }

1464                ASSERT(connp->conn_recv != NULL);
1465                ASSERT(connp->conn_recvicmp != NULL);

1467                IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1468                mutex_exit(&connfp->connf_lock);
1469                break;

1471        case IPPROTO_SCTP:
1472                /*
1473                 * The raw socket may have already been bound, remove it
1474                 * from the hash first.
1475                 */
1476                IPCL_HASH_REMOVE(connp);
1477                ret = ipcl_sctp_hash_insert(connp, lport);
1478                break;

1480        case IPPROTO_DCCP:
1481                cmn_err(CE_NOTE, "ipclassifier.c: ipcl_conn_insert_v4");

1483                connfp = &ipst->ips_ipcl_conn_fanout[
1484                    IPCL_CONN_HASH(connp->conn_faddr_v4,
1485                    connp->conn_ports, ipst)];
1486                mutex_enter(&connfp->connf_lock);

1488                IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1489                mutex_exit(&connfp->connf_lock);
1490                break;

1492 #endif /* ! codereview */
1493        default:
1494                /*
1495                 * Check for conflicts among MAC exempt bindings.  For
1496                 * transports with port numbers, this is done by the upper
1497                 * level per-transport binding logic.  For all others, it's
1498                 * done here.
1499                 */
1500                if (is_system_labeled() &&
1501                    check_exempt_conflict_v4(connp, ipst))
1502                        return (EADDRINUSE);
1503                /* FALLTHROUGH */

1505        case IPPROTO_UDP:
1506                if (protocol == IPPROTO_UDP) {
1507                        connfp = &ipst->ips_ipcl_udp_fanout[
1508                            IPCL_UDP_HASH(lport, ipst)];
1509                } else {
1510                        connfp = &ipst->ips_ipcl_proto_fanout_v4[protocol];
1511                }
```



```
1513                if (connp->conn_faddr_v4 != INADDR_ANY) {
1514                        IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1515                } else if (connp->conn_laddr_v4 != INADDR_ANY) {
1516                        IPCL_HASH_INSERT_BOUND(connfp, connp);
1517                } else {
1518                        IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1519                }
1520                break;
1521        }

1523        return (ret);
1524 }

1526 int
1527 ipcl_conn_insert_v6(conn_t *connp)
1528 {
1529        connf_t         *connfp;
1530        conn_t          *tconnp;
1531        int             ret = 0;
1532        ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
1533        uint16_t        lport = connp->conn_lport;
1534        uint8_t         protocol = connp->conn_proto;
1535        uint_t          ifindex = connp->conn_bound_if;

1537        if (IPCL_IS_IPTUN(connp))
1538                return (ipcl_iptun_hash_insert_v6(connp, ipst));

1540        switch (protocol) {
1541        case IPPROTO_TCP:

1543                /*
1544                 * For tcp, we check whether the connection tuple already
1545                 * exists before allowing the connection to proceed.  We
1546                 * also allow indexing on the zoneid. This is to allow
1547                 * multiple shared stack zones to have the same tcp
1548                 * connection tuple. In practice this only happens for
1549                 * ipv6_loopback as it's the only local address which
1550                 * doesn't have to be unique.
1551                 */
1552                connfp = &ipst->ips_ipcl_conn_fanout[
1553                    IPCL_CONN_HASH_V6(connp->conn_faddr_v6, connp->conn_ports,
1554                    ipst)];
1555                mutex_enter(&connfp->connf_lock);
1556                for (tconnp = connfp->connf_head; tconnp != NULL;
1557                    tconnp = tconnp->conn_next) {
1558                        /* NOTE: need to match zoneid. Bug in onnv-gate */
1559                        if (IPCL_CONN_MATCH_V6(tconnp, connp->conn_proto,
1560                            connp->conn_faddr_v6, connp->conn_laddr_v6,
1561                            connp->conn_ports) &&
1562                            (tconnp->conn_bound_if == 0 ||
1563                            tconnp->conn_bound_if == ifindex) &&
1564                            IPCL_ZONE_MATCH(tconnp, connp->conn_zoneid)) {
1565                                /* Already have a conn. bail out */
1566                                mutex_exit(&connfp->connf_lock);
1567                                return (EADDRINUSE);
1568                        }
1569                }
1570                if (connp->conn_fanout != NULL) {
1571                        /*
1572                         * Probably a XTI/TLI application trying to do a
1573                         * rebind. Let it happen.
1574                         */
1575                        mutex_exit(&connfp->connf_lock);
1576                        IPCL_HASH_REMOVE(connp);
1577                        mutex_enter(&connfp->connf_lock);
1578                }
```

```
1579                IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1580                mutex_exit(&connfp->connf_lock);
1581                break;

1583        case IPPROTO_SCTP:
1584                IPCL_HASH_REMOVE(connp);
1585                ret = ipcl_sctp_hash_insert(connp, lport);
1586                break;

1588        case IPPROTO_DCCP:
1589                /* XXX:DCCP */
1590                break;

1592 #endif /* ! codereview */
1593        default:
1594                if (is_system_labeled() &&
1595                    check_exempt_conflict_v6(connp, ipst))
1596                        return (EADDRINUSE);
1597                /* FALLTHROUGH */
1598        case IPPROTO_UDP:
1599                if (protocol == IPPROTO_UDP) {
1600                        connfp = &ipst->ips_ipcl_udp_fanout[
1601                            IPCL_UDP_HASH(lport, ipst)];
1602                } else {
1603                        connfp = &ipst->ips_ipcl_proto_fanout_v6[protocol];
1604                }

1606                if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6)) {
1607                        IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1608                } else if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
1609                        IPCL_HASH_INSERT_BOUND(connfp, connp);
1610                } else {
1611                        IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1612                }
1613                break;
1614        }

1616        return (ret);
1617 }

1619 /*
1620  * v4 packet classifying function. looks up the fanout table to
1621  * find the conn, the packet belongs to. returns the conn with
1622  * the reference held, null otherwise.
1623  *
1624  * If zoneid is ALL_ZONES, then the search rules described in the "Connection
1625  * Lookup" comment block are applied.  Labels are also checked as described
1626  * above.  If the packet is from the inside (looped back), and is from the same
1627  * zone, then label checks are omitted.
1628  */
1629 conn_t *
1630 ipcl_classify_v4(mblk_t *mp, uint8_t protocol, uint_t hdr_len,
1631     ip_recv_attr_t *ira, ip_stack_t *ipst)
1632 {
1633        ipha_t  *ipha;
1634        connf_t *connfp, *bind_connfp;
1635        uint16_t lport;
1636        uint16_t fport;
1637        uint32_t ports;
1638        conn_t  *connp;
1639        uint16_t  *up;
1640        zoneid_t        zoneid = ira->ira_zoneid;

1642        ipha = (ipha_t *)mp->b_rptr;
1643        up = (uint16_t *)((uchar_t *)ipha + hdr_len + TCP_PORTS_OFFSET);
```



```
1645        switch (protocol) {
1646        case IPPROTO_TCP:
1647                ports = *(uint32_t *)up;
1648                connfp =
1649                    &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ipha->ipha_src,
1650                    ports, ipst)];
1651                mutex_enter(&connfp->connf_lock);
1652                for (connp = connfp->connf_head; connp != NULL;
1653                    connp = connp->conn_next) {
1654                        if (IPCL_CONN_MATCH(connp, protocol,
1655                            ipha->ipha_src, ipha->ipha_dst, ports) &&
1656                            (connp->conn_zoneid == zoneid ||
1657                            connp->conn_allzones ||
1658                            ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1659                            (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1660                            (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1661                                break;
1662                }

1664                if (connp != NULL) {
1665                        /*
1666                         * We have a fully-bound TCP connection.
1667                         *
1668                         * For labeled systems, there's no need to check the
1669                         * label here.  It's known to be good as we checked
1670                         * before allowing the connection to become bound.
1671                         */
1672                        CONN_INC_REF(connp);
1673                        mutex_exit(&connfp->connf_lock);
1674                        return (connp);
1675                }

1677                mutex_exit(&connfp->connf_lock);
1678                lport = up[1];
1679                bind_connfp =
1680                    &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
1681                mutex_enter(&bind_connfp->connf_lock);
1682                for (connp = bind_connfp->connf_head; connp != NULL;
1683                    connp = connp->conn_next) {
1684                        if (IPCL_BIND_MATCH(connp, protocol, ipha->ipha_dst,
1685                            lport) &&
1686                            (connp->conn_zoneid == zoneid ||
1687                            connp->conn_allzones ||
1688                            ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1689                            (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1690                            (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1691                                break;
1692                }

1694                /*
1695                 * If the matching connection is SLP on a private address, then
1696                 * the label on the packet must match the local zone's label.
1697                 * Otherwise, it must be in the label range defined by tnrh.
1698                 * This is ensured by tsol_receive_local.
1699                 *
1700                 * Note that we don't check tsol_receive_local for
1701                 * the connected case.
1702                 */
1703                if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
1704                    !tsol_receive_local(mp, &ipha->ipha_dst, IPV4_VERSION,
1705                    ira, connp)) {
1706                        DTRACE_PROBE3(tx__ip__log__info__classify__tcp,
1707                            char *, "connp(1) could not receive mp(2)",
1708                            conn_t *, connp, mblk_t *, mp);
1709                        connp = NULL;
1710                }
```

```
1712                if (connp != NULL) {
1713                        /* Have a listener at least */
1714                        CONN_INC_REF(connp);
1715                        mutex_exit(&bind_connfp->connf_lock);
1716                        return (connp);
1717                }

1719                mutex_exit(&bind_connfp->connf_lock);
1720                break;

1722        case IPPROTO_UDP:
1723                lport = up[1];
1724                fport = up[0];
1725                connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
1726                mutex_enter(&connfp->connf_lock);
1727                for (connp = connfp->connf_head; connp != NULL;
1728                    connp = connp->conn_next) {
1729                        if (IPCL_UDP_MATCH(connp, lport, ipha->ipha_dst,
1730                            fport, ipha->ipha_src) &&
1731                            (connp->conn_zoneid == zoneid ||
1732                            connp->conn_allzones ||
1733                            ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1734                            (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE))))
1735                                break;
1736                }

1738                if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
1739                    !tsol_receive_local(mp, &ipha->ipha_dst, IPV4_VERSION,
1740                    ira, connp)) {
1741                        DTRACE_PROBE3(tx__ip__log__info__classify__udp,
1742                            char *, "connp(1) could not receive mp(2)",
1743                            conn_t *, connp, mblk_t *, mp);
1744                        connp = NULL;
1745                }

1747                if (connp != NULL) {
1748                        CONN_INC_REF(connp);
1749                        mutex_exit(&connfp->connf_lock);
1750                        return (connp);
1751                }

1753                /*
1754                 * We shouldn't come here for multicast/broadcast packets
1755                 */
1756                mutex_exit(&connfp->connf_lock);

1758                break;

1760        case IPPROTO_DCCP:
1761                fport = up[0];
1762                lport = up[1];
1763                connfp = &ipst->ips_ipcl_dccp_fanout[IPCL_DCCP_HASH(
1764                    lport, ipst)];
1765                mutex_enter(&connfp->connf_lock);
1766                for (connp = connfp->connf_head; connp != NULL;
1767                    connp = connp->conn_next) {
1768                        cmn_err(CE_NOTE, "connfp found");
1769                        /* XXX:DCCP */
1770                        if (IPCL_UDP_MATCH(connp, lport, ipha->ipha_dst,
1771                            fport, ipha->ipha_src)) {
1772                                break;
1773                        }
1774                }

1776                if (connp != NULL) {
```



```
1777                        CONN_INC_REF(connp);
1778                        mutex_exit(&connfp->connf_lock);
1779                        return (connp);
1780                }

1782                mutex_exit(&connfp->connf_lock);
1783                break;

1785 #endif /* ! codereview */
1786        case IPPROTO_ENCAP:
1787        case IPPROTO_IPV6:
1788                return (ipcl_iptun_classify_v4(&ipha->ipha_src,
1789                    &ipha->ipha_dst, ipst));
1790        }

1792        return (NULL);
1793 }

1795 conn_t *
1796 ipcl_classify_v6(mblk_t *mp, uint8_t protocol, uint_t hdr_len,
1797     ip_recv_attr_t *ira, ip_stack_t *ipst)
1798 {
1799        ip6_t           *ip6h;
1800        connf_t         *connfp, *bind_connfp;
1801        uint16_t        lport;
1802        uint16_t        fport;
1803        tcpha_t         *tcpha;
1804        uint32_t        ports;
1805        conn_t          *connp;
1806        uint16_t        *up;
1807        zoneid_t        zoneid = ira->ira_zoneid;

1809        ip6h = (ip6_t *)mp->b_rptr;

1811        switch (protocol) {
1812        case IPPROTO_TCP:
1813                tcpha = (tcpha_t *)&mp->b_rptr[hdr_len];
1814                up = &tcpha->tha_lport;
1815                ports = *(uint32_t *)up;

1817                connfp =
1818                    &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_src,
1819                    ports, ipst)];
1820                mutex_enter(&connfp->connf_lock);
1821                for (connp = connfp->connf_head; connp != NULL;
1822                    connp = connp->conn_next) {
1823                        if (IPCL_CONN_MATCH_V6(connp, protocol,
1824                            ip6h->ip6_src, ip6h->ip6_dst, ports) &&
1825                            (connp->conn_zoneid == zoneid ||
1826                            connp->conn_allzones ||
1827                            ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1828                            (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1829                            (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1830                                break;
1831                }

1833                if (connp != NULL) {
1834                        /*
1835                         * We have a fully-bound TCP connection.
1836                         *
1837                         * For labeled systems, there's no need to check the
1838                         * label here.  It's known to be good as we checked
1839                         * before allowing the connection to become bound.
1840                         */
1841                        CONN_INC_REF(connp);
1842                        mutex_exit(&connfp->connf_lock);
```

```
1843                        return (connp);
1844                }
1846                mutex_exit(&connfp->connf_lock);
1848                lport = up[1];
1849                bind_connfp =
1850                    &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
1851                mutex_enter(&bind_connfp->connf_lock);
1852                for (connp = bind_connfp->connf_head; connp != NULL;
1853                    connp = connp->conn_next) {
1854                        if (IPCL_BIND_MATCH_V6(connp, protocol,
1855                            ip6h->ip6_dst, lport) &&
1856                            (connp->conn_zoneid == zoneid ||
1857                            connp->conn_allzones ||
1858                            ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1859                            (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1860                            (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1861                                break;
1862                }
1864                if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
1865                    !tsol_receive_local(mp, &ip6h->ip6_dst, IPV6_VERSION,
1866                    ira, connp)) {
1867                        DTRACE_PROBE3(tx__ip__log__info__classify__tcp6,
1868                            char *, "connp(1) could not receive mp(2)",
1869                            conn_t *, connp, mblk_t *, mp);
1870                        connp = NULL;
1871                }
1873                if (connp != NULL) {
1874                        /* Have a listner at least */
1875                        CONN_INC_REF(connp);
1876                        mutex_exit(&bind_connfp->connf_lock);
1877                        return (connp);
1878                }
1880                mutex_exit(&bind_connfp->connf_lock);
1881                break;
1883        case IPPROTO_UDP:
1884                up = (uint16_t *)&mp->b_rptr[hdr_len];
1885                lport = up[1];
1886                fport = up[0];
1887                connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
1888                mutex_enter(&connfp->connf_lock);
1889                for (connp = connfp->connf_head; connp != NULL;
1890                    connp = connp->conn_next) {
1891                        if (IPCL_UDP_MATCH_V6(connp, lport, ip6h->ip6_dst,
1892                            fport, ip6h->ip6_src) &&
1893                            (connp->conn_zoneid == zoneid ||
1894                            connp->conn_allzones ||
1895                            ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1896                            (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1897                            (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1898                                break;
1899                }
1901                if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
1902                    !tsol_receive_local(mp, &ip6h->ip6_dst, IPV6_VERSION,
1903                    ira, connp)) {
1904                        DTRACE_PROBE3(tx__ip__log__info__classify__udp6,
1905                            char *, "connp(1) could not receive mp(2)",
1906                            conn_t *, connp, mblk_t *, mp);
1907                        connp = NULL;
1908                }
```



```
1910                if (connp != NULL) {
1911                        CONN_INC_REF(connp);
1912                        mutex_exit(&connfp->connf_lock);
1913                        return (connp);
1914                }
1916                /*
1917                 * We shouldn't come here for multicast/broadcast packets
1918                 */
1919                mutex_exit(&connfp->connf_lock);
1920                break;
1921        case IPPROTO_ENCAP:
1922        case IPPROTO_IPV6:
1923                return (ipcl_iptun_classify_v6(&ip6h->ip6_src,
1924                    &ip6h->ip6_dst, ipst));
1925        }
1927        return (NULL);
1928 }
1930 /*
1931  * wrapper around ipcl_classify_(v4,v6) routines.
1932  */
1933 conn_t *
1934 ipcl_classify(mblk_t *mp, ip_recv_attr_t *ira, ip_stack_t *ipst)
1935 {
1936        if (ira->ira_flags & IRAF_IS_IPV4) {
1937                return (ipcl_classify_v4(mp, ira->ira_protocol,
1938                    ira->ira_ip_hdr_length, ira, ipst));
1939        } else {
1940                return (ipcl_classify_v6(mp, ira->ira_protocol,
1941                    ira->ira_ip_hdr_length, ira, ipst));
1942        }
1943 }
1945 /*
1946  * Only used to classify SCTP RAW sockets
1947  */
1948 conn_t *
1949 ipcl_classify_raw(mblk_t *mp, uint8_t protocol, uint32_t ports,
1950     ipha_t *ipha, ip6_t *ip6h, ip_recv_attr_t *ira, ip_stack_t *ipst)
1951 {
1952        connf_t         *connfp;
1953        conn_t          *connp;
1954        in_port_t       lport;
1955        int             ipversion;
1956        const void      *dst;
1957        zoneid_t        zoneid = ira->ira_zoneid;
1959        lport = ((uint16_t *)&ports)[1];
1960        if (ira->ira_flags & IRAF_IS_IPV4) {
1961                dst = (const void *)&ipha->ipha_dst;
1962                ipversion = IPV4_VERSION;
1963        } else {
1964                dst = (const void *)&ip6h->ip6_dst;
1965                ipversion = IPV6_VERSION;
1966        }
1968        connfp = &ipst->ips_ipcl_raw_fanout[IPCL_RAW_HASH(ntohs(lport), ipst)];
1969        mutex_enter(&connfp->connf_lock);
1970        for (connp = connfp->connf_head; connp != NULL;
1971            connp = connp->conn_next) {
1972                /* We don't allow v4 fallback for v6 raw socket. */
1973                if (ipversion != connp->conn_ipversion)
1974                        continue;
```

```
1975                if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) &&
1976                    !IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
1977                        if (ipversion == IPV4_VERSION) {
1978                                if (!IPCL_CONN_MATCH(connp, protocol,
1979                                    ipha->ipha_src, ipha->ipha_dst, ports))
1980                                        continue;
1981                        } else {
1982                                if (!IPCL_CONN_MATCH_V6(connp, protocol,
1983                                    ip6h->ip6_src, ip6h->ip6_dst, ports))
1984                                        continue;
1985                        }
1986                } else {
1987                        if (ipversion == IPV4_VERSION) {
1988                                if (!IPCL_BIND_MATCH(connp, protocol,
1989                                    ipha->ipha_dst, lport))
1990                                        continue;
1991                        } else {
1992                                if (!IPCL_BIND_MATCH_V6(connp, protocol,
1993                                    ip6h->ip6_dst, lport))
1994                                        continue;
1995                        }
1996                }

1998                if (connp->conn_zoneid == zoneid ||
1999                    connp->conn_allzones ||
2000                    ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
2001                    (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
2002                    (ira->ira_flags & IRAF_TX_SHARED_ADDR)))
2003                        break;
2004        }

2006        if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2007            !tsol_receive_local(mp, dst, ipversion, ira, connp)) {
2008                DTRACE_PROBE3(tx__ip__log__info__classify__rawip,
2009                    char *, "connp(1) could not receive mp(2)",
2010                    conn_t *, connp, mblk_t *, mp);
2011                connp = NULL;
2012        }

2014        if (connp != NULL)
2015                goto found;
2016        mutex_exit(&connfp->connf_lock);

2018        /* Try to look for a wildcard SCTP RAW socket match. */
2019        connfp = &ipst->ips_ipcl_raw_fanout[IPCL_RAW_HASH(0, ipst)];
2020        mutex_enter(&connfp->connf_lock);
2021        for (connp = connfp->connf_head; connp != NULL;
2022            connp = connp->conn_next) {
2023                /* We don't allow v4 fallback for v6 raw socket. */
2024                if (ipversion != connp->conn_ipversion)
2025                        continue;
2026                if (!IPCL_ZONE_MATCH(connp, zoneid))
2027                        continue;

2029                if (ipversion == IPV4_VERSION) {
2030                        if (IPCL_RAW_MATCH(connp, protocol, ipha->ipha_dst))
2031                                break;
2032                } else {
2033                        if (IPCL_RAW_MATCH_V6(connp, protocol, ip6h->ip6_dst)) {
2034                                break;
2035                        }
2036                }
2037        }

2039        if (connp != NULL)
2040                goto found;
```



```
2042        mutex_exit(&connfp->connf_lock);
2043        return (NULL);

2045 found:
2046        ASSERT(connp != NULL);
2047        CONN_INC_REF(connp);
2048        mutex_exit(&connfp->connf_lock);
2049        return (connp);
2050 }

2052 /* ARGSUSED */
2053 static int
2054 tcp_conn_constructor(void *buf, void *cdrarg, int kmflags)
2055 {
2056        itc_t   *itc = (itc_t *)buf;
2057        conn_t  *connp = &itc->itc_conn;
2058        tcp_t   *tcp = (tcp_t *)&itc[1];

2060        bzero(connp, sizeof (conn_t));
2061        bzero(tcp, sizeof (tcp_t));

2063        mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2064        cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2065        cv_init(&connp->conn_sq_cv, NULL, CV_DEFAULT, NULL);
2066        tcp->tcp_timercache = tcp_timermp_alloc(kmflags);
2067        if (tcp->tcp_timercache == NULL)
2068                return (ENOMEM);
2069        connp->conn_tcp = tcp;
2070        connp->conn_flags = IPCL_TCPCONN;
2071        connp->conn_proto = IPPROTO_TCP;
2072        tcp->tcp_connp = connp;
2073        rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);

2075        connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2076        if (connp->conn_ixa == NULL) {
2077                tcp_timermp_free(tcp);
2078                return (ENOMEM);
2079        }
2080        connp->conn_ixa->ixa_refcnt = 1;
2081        connp->conn_ixa->ixa_protocol = connp->conn_proto;
2082        connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2083        return (0);
2084 }

2086 /* ARGSUSED */
2087 static void
2088 tcp_conn_destructor(void *buf, void *cdrarg)
2089 {
2090        itc_t   *itc = (itc_t *)buf;
2091        conn_t  *connp = &itc->itc_conn;
2092        tcp_t   *tcp = (tcp_t *)&itc[1];

2094        ASSERT(connp->conn_flags & IPCL_TCPCONN);
2095        ASSERT(tcp->tcp_connp == connp);
2096        ASSERT(connp->conn_tcp == tcp);
2097        tcp_timermp_free(tcp);
2098        mutex_destroy(&connp->conn_lock);
2099        cv_destroy(&connp->conn_cv);
2100        cv_destroy(&connp->conn_sq_cv);
2101        rw_destroy(&connp->conn_ilg_lock);

2103        /* Can be NULL if constructor failed */
2104        if (connp->conn_ixa != NULL) {
2105                ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2106                ASSERT(connp->conn_ixa->ixa_ire == NULL);
```

```
2107                 ASSERT(connp->conn_ixa->ixa_nce == NULL);
2108                 ixa_refrele(connp->conn_ixa);
2109         }
2110 }

2112 /* ARGSUSED */
2113 static int
2114 ip_conn_constructor(void *buf, void *cdrarg, int kmflags)
2115 {
2116         itc_t   *itc = (itc_t *)buf;
2117         conn_t  *connp = &itc->itc_conn;

2119         bzero(connp, sizeof (conn_t));
2120         mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2121         cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2122         connp->conn_flags = IPCL_IPCCONN;
2123         rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);

2125         connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2126         if (connp->conn_ixa == NULL)
2127                 return (ENOMEM);
2128         connp->conn_ixa->ixa_refcnt = 1;
2129         connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2130         return (0);
2131 }

2133 /* ARGSUSED */
2134 static void
2135 ip_conn_destructor(void *buf, void *cdrarg)
2136 {
2137         itc_t   *itc = (itc_t *)buf;
2138         conn_t  *connp = &itc->itc_conn;

2140         ASSERT(connp->conn_flags & IPCL_IPCCONN);
2141         ASSERT(connp->conn_priv == NULL);
2142         mutex_destroy(&connp->conn_lock);
2143         cv_destroy(&connp->conn_cv);
2144         rw_destroy(&connp->conn_ilg_lock);

2146         /* Can be NULL if constructor failed */
2147         if (connp->conn_ixa != NULL) {
2148                 ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2149                 ASSERT(connp->conn_ixa->ixa_ire == NULL);
2150                 ASSERT(connp->conn_ixa->ixa_nce == NULL);
2151                 ixa_refrele(connp->conn_ixa);
2152         }
2153 }

2155 /* ARGSUSED */
2156 static int
2157 udp_conn_constructor(void *buf, void *cdrarg, int kmflags)
2158 {
2159         itc_t   *itc = (itc_t *)buf;
2160         conn_t  *connp = &itc->itc_conn;
2161         udp_t   *udp = (udp_t *)&itc[1];

2163         bzero(connp, sizeof (conn_t));
2164         bzero(udp, sizeof (udp_t));

2166         mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2167         cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2168         connp->conn_udp = udp;
2169         connp->conn_flags = IPCL_UDPCONN;
2170         connp->conn_proto = IPPROTO_UDP;
2171         udp->udp_connp = connp;
2172         rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);
```



```
2173         connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2174         if (connp->conn_ixa == NULL)
2175                 return (ENOMEM);
2176         connp->conn_ixa->ixa_refcnt = 1;
2177         connp->conn_ixa->ixa_protocol = connp->conn_proto;
2178         connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2179         return (0);
2180 }

2182 /* ARGSUSED */
2183 static void
2184 udp_conn_destructor(void *buf, void *cdrarg)
2185 {
2186         itc_t   *itc = (itc_t *)buf;
2187         conn_t  *connp = &itc->itc_conn;
2188         udp_t   *udp = (udp_t *)&itc[1];

2190         ASSERT(connp->conn_flags & IPCL_UDPCONN);
2191         ASSERT(udp->udp_connp == connp);
2192         ASSERT(connp->conn_udp == udp);
2193         mutex_destroy(&connp->conn_lock);
2194         cv_destroy(&connp->conn_cv);
2195         rw_destroy(&connp->conn_ilg_lock);

2197         /* Can be NULL if constructor failed */
2198         if (connp->conn_ixa != NULL) {
2199                 ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2200                 ASSERT(connp->conn_ixa->ixa_ire == NULL);
2201                 ASSERT(connp->conn_ixa->ixa_nce == NULL);
2202                 ixa_refrele(connp->conn_ixa);
2203         }
2204 }

2206 /* ARGSUSED */
2207 static int
2208 rawip_conn_constructor(void *buf, void *cdrarg, int kmflags)
2209 {
2210         itc_t   *itc = (itc_t *)buf;
2211         conn_t  *connp = &itc->itc_conn;
2212         icmp_t  *icmp = (icmp_t *)&itc[1];

2214         bzero(connp, sizeof (conn_t));
2215         bzero(icmp, sizeof (icmp_t));

2217         mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2218         cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2219         connp->conn_icmp = icmp;
2220         connp->conn_flags = IPCL_RAWIPCONN;
2221         connp->conn_proto = IPPROTO_ICMP;
2222         icmp->icmp_connp = connp;
2223         rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);
2224         connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2225         if (connp->conn_ixa == NULL)
2226                 return (ENOMEM);
2227         connp->conn_ixa->ixa_refcnt = 1;
2228         connp->conn_ixa->ixa_protocol = connp->conn_proto;
2229         connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2230         return (0);
2231 }

2233 /* ARGSUSED */
2234 static void
2235 rawip_conn_destructor(void *buf, void *cdrarg)
2236 {
2237         itc_t   *itc = (itc_t *)buf;
2238         conn_t  *connp = &itc->itc_conn;
```

```
2239         icmp_t  *icmp = (icmp_t *)&itc[1];

2241         ASSERT(connp->conn_flags & IPCL_RAWIPCONN);
2242         ASSERT(icmp->icmp_connp == connp);
2243         ASSERT(connp->conn_icmp == icmp);
2244         mutex_destroy(&connp->conn_lock);
2245         cv_destroy(&connp->conn_cv);
2246         rw_destroy(&connp->conn_ilg_lock);

2248         /* Can be NULL if constructor failed */
2249         if (connp->conn_ixa != NULL) {
2250                 ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2251                 ASSERT(connp->conn_ixa->ixa_ire == NULL);
2252                 ASSERT(connp->conn_ixa->ixa_nce == NULL);
2253                 ixa_refrele(connp->conn_ixa);
2254         }
2255 }

2257 /* ARGSUSED */
2258 static int
2259 rts_conn_constructor(void *buf, void *cdrarg, int kmflags)
2260 {
2261         itc_t   *itc = (itc_t *)buf;
2262         conn_t  *connp = &itc->itc_conn;
2263         rts_t   *rts = (rts_t *)&itc[1];

2265         bzero(connp, sizeof (conn_t));
2266         bzero(rts, sizeof (rts_t));

2268         mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2269         cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2270         connp->conn_rts = rts;
2271         connp->conn_flags = IPCL_RTSCONN;
2272         rts->rts_connp = connp;
2273         rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);
2274         connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2275         if (connp->conn_ixa == NULL)
2276                 return (ENOMEM);
2277         connp->conn_ixa->ixa_refcnt = 1;
2278         connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2279         return (0);
2280 }

2282 /* ARGSUSED */
2283 static void
2284 rts_conn_destructor(void *buf, void *cdrarg)
2285 {
2286         itc_t   *itc = (itc_t *)buf;
2287         conn_t  *connp = &itc->itc_conn;
2288         rts_t   *rts = (rts_t *)&itc[1];

2290         ASSERT(connp->conn_flags & IPCL_RTSCONN);
2291         ASSERT(rts->rts_connp == connp);
2292         ASSERT(connp->conn_rts == rts);
2293         mutex_destroy(&connp->conn_lock);
2294         cv_destroy(&connp->conn_cv);
2295         rw_destroy(&connp->conn_ilg_lock);

2297         /* Can be NULL if constructor failed */
2298         if (connp->conn_ixa != NULL) {
2299                 ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2300                 ASSERT(connp->conn_ixa->ixa_ire == NULL);
2301                 ASSERT(connp->conn_ixa->ixa_nce == NULL);
2302                 ixa_refrele(connp->conn_ixa);
2303         }
2304 }
```



```
2306 /* ARGSUSED */
2307 static int
2308 dccp_conn_constructor(void *buf, void *cdrarg, int kmflags)
2309 {
2310         itc_t   *itc = (itc_t *)buf;
2311         conn_t  *connp = &itc->itc_conn;
2312         dccp_t  *dccp = (dccp_t *)&itc[1];

2314         bzero(connp, sizeof (conn_t));
2315         bzero(dccp, sizeof (dccp_t));

2317         mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2318         cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2319         rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);

2321         connp->conn_dccp = dccp;
2322         connp->conn_flags = IPCL_DCCPCONN;
2323         connp->conn_proto = IPPROTO_DCCP;
2324         dccp->dccp_connp = connp;
2325         connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2326         if (connp->conn_ixa == NULL) {
2327                 return (NULL);
2328         }
2329         connp->conn_ixa->ixa_refcnt = 1;
2330         connp->conn_ixa->ixa_protocol = connp->conn_proto;
2331         connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);

2333         return (0);
2334 }

2336 /* ARGSUSED */
2337 static void
2338 dccp_conn_destructor(void *buf, void *cdrarg)
2339 {
2340         itc_t   *itc = (itc_t *)buf;
2341         conn_t  *connp = &itc->itc_conn;
2342         dccp_t  *dccp = (dccp_t *)&itc[1];

2344         ASSERT(connp->conn_flags & IPCL_DCCPCONN);
2345         ASSERT(dccp->dccp_connp == connp);
2346         ASSERT(connp->conn_dccp == dccp);

2348         mutex_destroy(&connp->conn_lock);
2349         cv_destroy(&connp->conn_cv);
2350         rw_destroy(&connp->conn_ilg_lock);

2352         if (connp->conn_ixa != NULL) {
2353                 ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2354                 ASSERT(connp->conn_ixa->ixa_ire == NULL);
2355                 ASSERT(connp->conn_ixa->ixa_nce == NULL);

2357                 ixa_refrele(connp->conn_ixa);
2358         }
2359 }

2361 #endif /* ! codereview */
2362 /*
2363  * Called as part of ipcl_conn_destroy to assert and clear any pointers
2364  * in the conn_t.
2365  *
2366  * Below we list all the pointers in the conn_t as a documentation aid.
2367  * The ones that we can not ASSERT to be NULL are #ifdef'ed out.
2368  * If you add any pointers to the conn_t please add an ASSERT here
2369  * and #ifdef it out if it can't be actually asserted to be NULL.
2370  * In any case, we bzero most of the conn_t at the end of the function.
```

```
 */
void
ipcl_conn_cleanup(conn_t *connp)
{
        ip_xmit_attr_t  *ixa;

        ASSERT(connp->conn_latch == NULL);
        ASSERT(connp->conn_latch_in_policy == NULL);
        ASSERT(connp->conn_latch_in_action == NULL);
#ifdef notdef
        ASSERT(connp->conn_rq == NULL);
        ASSERT(connp->conn_wq == NULL);
#endif
        ASSERT(connp->conn_cred == NULL);
        ASSERT(connp->conn_g_fanout == NULL);
        ASSERT(connp->conn_g_next == NULL);
        ASSERT(connp->conn_g_prev == NULL);
        ASSERT(connp->conn_policy == NULL);
        ASSERT(connp->conn_fanout == NULL);
        ASSERT(connp->conn_next == NULL);
        ASSERT(connp->conn_prev == NULL);
        ASSERT(connp->conn_oper_pending_ill == NULL);
        ASSERT(connp->conn_ilg == NULL);
        ASSERT(connp->conn_drain_next == NULL);
        ASSERT(connp->conn_drain_prev == NULL);
#ifdef notdef
        /* conn_idl is not cleared when removed from idl list */
        ASSERT(connp->conn_idl == NULL);
#endif
        ASSERT(connp->conn_ipsec_opt_mp == NULL);
#ifdef notdef
        /* conn_netstack is cleared by the caller; needed by ixa_cleanup */
        ASSERT(connp->conn_netstack == NULL);
#endif

        ASSERT(connp->conn_helper_info == NULL);
        ASSERT(connp->conn_ixa != NULL);
        ixa = connp->conn_ixa;
        ASSERT(ixa->ixa_refcnt == 1);
        /* Need to preserve ixa_protocol */
        ixa_cleanup(ixa);
        ixa->ixa_flags = 0;

        /* Clear out the conn_t fields that are not preserved */
        bzero(&connp->conn_start_clr,
            sizeof (conn_t) -
            ((uchar_t *)&connp->conn_start_clr - (uchar_t *)connp));
}

/*
 * All conns are inserted in a global multi-list for the benefit of
 * walkers. The walk is guaranteed to walk all open conns at the time
 * of the start of the walk exactly once. This property is needed to
 * achieve some cleanups during unplumb of interfaces. This is achieved
 * as follows.
 *
 * ipcl_conn_create and ipcl_conn_destroy are the only functions that
 * call the insert and delete functions below at creation and deletion
 * time respectively. The conn never moves or changes its position in this
 * multi-list during its lifetime. CONN_CONDEMNED ensures that the refcnt
 * won't increase due to walkers, once the conn deletion has started. Note
 * that we can't remove the conn from the global list and then wait for
 * the refcnt to drop to zero, since walkers would then see a truncated
 * list. CONN_INCIPIENT ensures that walkers don't start looking at
 * conns until ip_open is ready to make them globally visible.
 * The global round robin multi-list locks are held only to get the
```



```
 * next member/insertion/deletion and contention should be negligible
 * if the multi-list is much greater than the number of cpus.
 */
void
ipcl_globalhash_insert(conn_t *connp)
{
        int     index;
        struct connf_s  *connfp;
        ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;

        /*
         * No need for atomic here. Approximate even distribution
         * in the global lists is sufficient.
         */
        ipst->ips_conn_g_index++;
        index = ipst->ips_conn_g_index & (CONN_G_HASH_SIZE - 1);

        connp->conn_g_prev = NULL;
        /*
         * Mark as INCIPIENT, so that walkers will ignore this
         * for now, till ip_open is ready to make it visible globally.
         */
        connp->conn_state_flags |= CONN_INCIPIENT;

        connfp = &ipst->ips_ipcl_globalhash_fanout[index];
        /* Insert at the head of the list */
        mutex_enter(&connfp->connf_lock);
        connp->conn_g_next = connfp->connf_head;
        if (connp->conn_g_next != NULL)
                connp->conn_g_next->conn_g_prev = connp;
        connfp->connf_head = connp;

        /* The fanout bucket this conn points to */
        connp->conn_g_fanout = connfp;

        mutex_exit(&connfp->connf_lock);
}

void
ipcl_globalhash_remove(conn_t *connp)
{
        struct connf_s  *connfp;

        /*
         * We were never inserted in the global multi list.
         * IPCL_NONE variety is never inserted in the global multilist
         * since it is presumed to not need any cleanup and is transient.
         */
        if (connp->conn_g_fanout == NULL)
                return;

        connfp = connp->conn_g_fanout;
        mutex_enter(&connfp->connf_lock);
        if (connp->conn_g_prev != NULL)
                connp->conn_g_prev->conn_g_next = connp->conn_g_next;
        else
                connfp->connf_head = connp->conn_g_next;
        if (connp->conn_g_next != NULL)
                connp->conn_g_next->conn_g_prev = connp->conn_g_prev;
        mutex_exit(&connfp->connf_lock);

        /* Better to stumble on a null pointer than to corrupt memory */
        connp->conn_g_next = NULL;
        connp->conn_g_prev = NULL;
        connp->conn_g_fanout = NULL;
}
```

```
2504 /*
2505  * Walk the list of all conn_t's in the system, calling the function provided
2506  * With the specified argument for each.
2507  * Applies to both IPv4 and IPv6.
2508  *
2509  * CONNs may hold pointers to ills (conn_dhcpinit_ill and
2510  * conn_oper_pending_ill). To guard against stale pointers
2511  * ipcl_walk() is called to cleanup the conn_t's, typically when an interface is
2512  * unplumbed or removed. New conn_t's that are created while we are walking
2513  * may be missed by this walk, because they are not necessarily inserted
2514  * at the tail of the list. They are new conn_t's and thus don't have any
2515  * stale pointers. The CONN_CLOSING flag ensures that no new reference
2516  * is created to the struct that is going away.
2517  */
2518 void
2519 ipcl_walk(pfv_t func, void *arg, ip_stack_t *ipst)
2520 {
2521         int     i;
2522         conn_t  *connp;
2523         conn_t  *prev_connp;

2525         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
2526                 mutex_enter(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2527                 prev_connp = NULL;
2528                 connp = ipst->ips_ipcl_globalhash_fanout[i].connf_head;
2529                 while (connp != NULL) {
2530                         mutex_enter(&connp->conn_lock);
2531                         if (connp->conn_state_flags &
2532                             (CONN_CONDEMNED | CONN_INCIPIENT)) {
2533                                 mutex_exit(&connp->conn_lock);
2534                                 connp = connp->conn_g_next;
2535                                 continue;
2536                         }
2537                         CONN_INC_REF_LOCKED(connp);
2538                         mutex_exit(&connp->conn_lock);
2539                         mutex_exit(
2540                             &ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2541                         (*func)(connp, arg);
2542                         if (prev_connp != NULL)
2543                                 CONN_DEC_REF(prev_connp);
2544                         mutex_enter(
2545                             &ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2546                         prev_connp = connp;
2547                         connp = connp->conn_g_next;
2548                 }
2549                 mutex_exit(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2550                 if (prev_connp != NULL)
2551                         CONN_DEC_REF(prev_connp);
2552         }
2553 }

2555 /*
2556  * Search for a peer TCP/IPv4 loopback conn by doing a reverse lookup on
2557  * the {src, dst, lport, fport} quadruplet.  Returns with conn reference
2558  * held; caller must call CONN_DEC_REF.  Only checks for connected entries
2559  * (peer tcp in ESTABLISHED state).
2560  */
2561 conn_t *
2562 ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *connp, ipha_t *ipha, tcpha_t *tcpha,
2563     ip_stack_t *ipst)
2564 {
2565         uint32_t ports;
2566         uint16_t *pports = (uint16_t *)&ports;
2567         connf_t *connfp;
2568         conn_t  *tconnp;
```



```
2569         boolean_t zone_chk;

2571         /*
2572          * If either the source of destination address is loopback, then
2573          * both endpoints must be in the same Zone.  Otherwise, both of
2574          * the addresses are system-wide unique (tcp is in ESTABLISHED
2575          * state) and the endpoints may reside in different Zones.
2576          */
2577         zone_chk = (ipha->ipha_src == htonl(INADDR_LOOPBACK) ||
2578             ipha->ipha_dst == htonl(INADDR_LOOPBACK));

2580         pports[0] = tcpha->tha_fport;
2581         pports[1] = tcpha->tha_lport;

2583         connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ipha->ipha_dst,
2584             ports, ipst)];

2586         mutex_enter(&connfp->connf_lock);
2587         for (tconnp = connfp->connf_head; tconnp != NULL;
2588             tconnp = tconnp->conn_next) {

2590                 if (IPCL_CONN_MATCH(tconnp, IPPROTO_TCP,
2591                     ipha->ipha_dst, ipha->ipha_src, ports) &&
2592                     tconnp->conn_tcp->tcp_state == TCPS_ESTABLISHED &&
2593                     (!zone_chk || tconnp->conn_zoneid == connp->conn_zoneid)) {

2595                         ASSERT(tconnp != connp);
2596                         CONN_INC_REF(tconnp);
2597                         mutex_exit(&connfp->connf_lock);
2598                         return (tconnp);
2599                 }
2600         }
2601         mutex_exit(&connfp->connf_lock);
2602         return (NULL);
2603 }

2605 /*
2606  * Search for a peer TCP/IPv6 loopback conn by doing a reverse lookup on
2607  * the {src, dst, lport, fport} quadruplet.  Returns with conn reference
2608  * held; caller must call CONN_DEC_REF.  Only checks for connected entries
2609  * (peer tcp in ESTABLISHED state).
2610  */
2611 conn_t *
2612 ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *connp, ip6_t *ip6h, tcpha_t *tcpha,
2613     ip_stack_t *ipst)
2614 {
2615         uint32_t ports;
2616         uint16_t *pports = (uint16_t *)&ports;
2617         connf_t *connfp;
2618         conn_t  *tconnp;
2619         boolean_t zone_chk;

2621         /*
2622          * If either the source of destination address is loopback, then
2623          * both endpoints must be in the same Zone.  Otherwise, both of
2624          * the addresses are system-wide unique (tcp is in ESTABLISHED
2625          * state) and the endpoints may reside in different Zones.  We
2626          * don't do Zone check for link local address(es) because the
2627          * current Zone implementation treats each link local address as
2628          * being unique per system node, i.e. they belong to global Zone.
2629          */
2630         zone_chk = (IN6_IS_ADDR_LOOPBACK(&ip6h->ip6_src) ||
2631             IN6_IS_ADDR_LOOPBACK(&ip6h->ip6_dst));

2633         pports[0] = tcpha->tha_fport;
2634         pports[1] = tcpha->tha_lport;
```

```
2636         connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_dst,
2637             ports, ipst)];

2639         mutex_enter(&connfp->connf_lock);
2640         for (tconnp = connfp->connf_head; tconnp != NULL;
2641             tconnp = tconnp->conn_next) {

2643                 /* We skip conn_bound_if check here as this is loopback tcp */
2644                 if (IPCL_CONN_MATCH_V6(tconnp, IPPROTO_TCP,
2645                     ip6h->ip6_dst, ip6h->ip6_src, ports) &&
2646                     tconnp->conn_tcp->tcp_state == TCPS_ESTABLISHED &&
2647                     (!zone_chk || tconnp->conn_zoneid == connp->conn_zoneid)) {

2649                         ASSERT(tconnp != connp);
2650                         CONN_INC_REF(tconnp);
2651                         mutex_exit(&connfp->connf_lock);
2652                         return (tconnp);
2653                 }
2654         }
2655         mutex_exit(&connfp->connf_lock);
2656         return (NULL);
2657 }

2659 /*
2660  * Find an exact {src, dst, lport, fport} match for a bounced datagram.
2661  * Returns with conn reference held. Caller must call CONN_DEC_REF.
2662  * Only checks for connected entries i.e. no INADDR_ANY checks.
2663  */
2664 conn_t *
2665 ipcl_tcp_lookup_reversed_ipv4(ipha_t *ipha, tcpha_t *tcpha, int min_state,
2666     ip_stack_t *ipst)
2667 {
2668         uint32_t ports;
2669         uint16_t *pports;
2670         connf_t *connfp;
2671         conn_t  *tconnp;

2673         pports = (uint16_t *)&ports;
2674         pports[0] = tcpha->tha_fport;
2675         pports[1] = tcpha->tha_lport;

2677         connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ipha->ipha_dst,
2678             ports, ipst)];

2680         mutex_enter(&connfp->connf_lock);
2681         for (tconnp = connfp->connf_head; tconnp != NULL;
2682             tconnp = tconnp->conn_next) {

2684                 if (IPCL_CONN_MATCH(tconnp, IPPROTO_TCP,
2685                     ipha->ipha_dst, ipha->ipha_src, ports) &&
2686                     tconnp->conn_tcp->tcp_state >= min_state) {

2688                         CONN_INC_REF(tconnp);
2689                         mutex_exit(&connfp->connf_lock);
2690                         return (tconnp);
2691                 }
2692         }
2693         mutex_exit(&connfp->connf_lock);
2694         return (NULL);
2695 }

2697 /*
2698  * Find an exact {src, dst, lport, fport} match for a bounced datagram.
2699  * Returns with conn reference held. Caller must call CONN_DEC_REF.
2700  * Only checks for connected entries i.e. no INADDR_ANY checks.
```



```
2701  * Match on ifindex in addition to addresses.
2702  */
2703 conn_t *
2704 ipcl_tcp_lookup_reversed_ipv6(ip6_t *ip6h, tcpha_t *tcpha, int min_state,
2705     uint_t ifindex, ip_stack_t *ipst)
2706 {
2707         tcp_t   *tcp;
2708         uint32_t ports;
2709         uint16_t *pports;
2710         connf_t *connfp;
2711         conn_t  *tconnp;

2713         pports = (uint16_t *)&ports;
2714         pports[0] = tcpha->tha_fport;
2715         pports[1] = tcpha->tha_lport;

2717         connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_dst,
2718             ports, ipst)];

2720         mutex_enter(&connfp->connf_lock);
2721         for (tconnp = connfp->connf_head; tconnp != NULL;
2722             tconnp = tconnp->conn_next) {

2724                 tcp = tconnp->conn_tcp;
2725                 if (IPCL_CONN_MATCH_V6(tconnp, IPPROTO_TCP,
2726                     ip6h->ip6_dst, ip6h->ip6_src, ports) &&
2727                     tcp->tcp_state >= min_state &&
2728                     (tconnp->conn_bound_if == 0 ||
2729                     tconnp->conn_bound_if == ifindex)) {

2731                         CONN_INC_REF(tconnp);
2732                         mutex_exit(&connfp->connf_lock);
2733                         return (tconnp);
2734                 }
2735         }
2736         mutex_exit(&connfp->connf_lock);
2737         return (NULL);
2738 }

2740 /*
2741  * Finds a TCP/IPv4 listening connection; called by tcp_disconnect to locate
2742  * a listener when changing state.
2743  */
2744 conn_t *
2745 ipcl_lookup_listener_v4(uint16_t lport, ipaddr_t laddr, zoneid_t zoneid,
2746     ip_stack_t *ipst)
2747 {
2748         connf_t         *bind_connfp;
2749         conn_t          *connp;
2750         tcp_t           *tcp;

2752         /*
2753          * Avoid false matches for packets sent to an IP destination of
2754          * all zeros.
2755          */
2756         if (laddr == 0)
2757                 return (NULL);

2759         ASSERT(zoneid != ALL_ZONES);

2761         bind_connfp = &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
2762         mutex_enter(&bind_connfp->connf_lock);
2763         for (connp = bind_connfp->connf_head; connp != NULL;
2764             connp = connp->conn_next) {
2765                 tcp = connp->conn_tcp;
2766                 if (IPCL_BIND_MATCH(connp, IPPROTO_TCP, laddr, lport) &&
```

```
2767                    IPCL_ZONE_MATCH(connp, zoneid) &&
2768                    (tcp->tcp_listener == NULL)) {
2769                        CONN_INC_REF(connp);
2770                        mutex_exit(&bind_connfp->connf_lock);
2771                        return (connp);
2772                }
2773        }
2774        mutex_exit(&bind_connfp->connf_lock);
2775        return (NULL);
2776 }

2778 /*
2779  * Finds a TCP/IPv6 listening connection; called by tcp_disconnect to locate
2780  * a listener when changing state.
2781  */
2782 conn_t *
2783 ipcl_lookup_listener_v6(uint16_t lport, in6_addr_t *laddr, uint_t ifindex,
2784     zoneid_t zoneid, ip_stack_t *ipst)
2785 {
2786        connf_t         *bind_connfp;
2787        conn_t          *connp = NULL;
2788        tcp_t           *tcp;

2790        /*
2791         * Avoid false matches for packets sent to an IP destination of
2792         * all zeros.
2793         */
2794        if (IN6_IS_ADDR_UNSPECIFIED(laddr))
2795                return (NULL);

2797        ASSERT(zoneid != ALL_ZONES);

2799        bind_connfp = &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
2800        mutex_enter(&bind_connfp->connf_lock);
2801        for (connp = bind_connfp->connf_head; connp != NULL;
2802            connp = connp->conn_next) {
2803                tcp = connp->conn_tcp;
2804                if (IPCL_BIND_MATCH_V6(connp, IPPROTO_TCP, *laddr, lport) &&
2805                    IPCL_ZONE_MATCH(connp, zoneid) &&
2806                    (connp->conn_bound_if == 0 ||
2807                    connp->conn_bound_if == ifindex) &&
2808                    tcp->tcp_listener == NULL) {
2809                        CONN_INC_REF(connp);
2810                        mutex_exit(&bind_connfp->connf_lock);
2811                        return (connp);
2812                }
2813        }
2814        mutex_exit(&bind_connfp->connf_lock);
2815        return (NULL);
2816 }

2818 /*
2819  * ipcl_get_next_conn
2820  *      get the next entry in the conn global list
2821  *      and put a reference on the next_conn.
2822  *      decrement the reference on the current conn.
2823  *
2824  * This is an iterator based walker function that also provides for
2825  * some selection by the caller. It walks through the conn_hash bucket
2826  * searching for the next valid connp in the list, and selects connections
2827  * that are neither closed nor condemned. It also REFHOLDS the conn
2828  * thus ensuring that the conn exists when the caller uses the conn.
2829  */
2830 conn_t *
2831 ipcl_get_next_conn(connf_t *connfp, conn_t *connp, uint32_t conn_flags)
2832 {
```



```
2833        conn_t  *next_connp;

2835        if (connfp == NULL)
2836                return (NULL);

2838        mutex_enter(&connfp->connf_lock);

2840        next_connp = (connp == NULL) ?
2841            connfp->connf_head : connp->conn_g_next;

2843        while (next_connp != NULL) {
2844                mutex_enter(&next_connp->conn_lock);
2845                if (!(next_connp->conn_flags & conn_flags) ||
2846                    (next_connp->conn_state_flags &
2847                    (CONN_CONDEMNED | CONN_INCIPIENT))) {
2848                        /*
2849                         * This conn has been condemned or
2850                         * is closing, or the flags don't match
2851                         */
2852                        mutex_exit(&next_connp->conn_lock);
2853                        next_connp = next_connp->conn_g_next;
2854                        continue;
2855                }
2856                CONN_INC_REF_LOCKED(next_connp);
2857                mutex_exit(&next_connp->conn_lock);
2858                break;
2859        }

2861        mutex_exit(&connfp->connf_lock);

2863        if (connp != NULL)
2864                CONN_DEC_REF(connp);

2866        return (next_connp);
2867 }

2869 #ifdef CONN_DEBUG
2870 /*
2871  * Trace of the last NBUF refhold/refrele
2872  */
2873 int
2874 conn_trace_ref(conn_t *connp)
2875 {
2876        int     last;
2877        conn_trace_t    *ctb;

2879        ASSERT(MUTEX_HELD(&connp->conn_lock));
2880        last = connp->conn_trace_last;
2881        last++;
2882        if (last == CONN_TRACE_MAX)
2883                last = 0;

2885        ctb = &connp->conn_trace_buf[last];
2886        ctb->ctb_depth = getpcstack(ctb->ctb_stack, CONN_STACK_DEPTH);
2887        connp->conn_trace_last = last;
2888        return (1);
2889 }

2891 int
2892 conn_untrace_ref(conn_t *connp)
2893 {
2894        int     last;
2895        conn_trace_t    *ctb;

2897        ASSERT(MUTEX_HELD(&connp->conn_lock));
2898        last = connp->conn_trace_last;
```

```
2899         last++;
2900         if (last == CONN_TRACE_MAX)
2901                 last = 0;

2903         ctb = &connp->conn_trace_buf[last];
2904         ctb->ctb_depth = getpcstack(ctb->ctb_stack, CONN_STACK_DEPTH);
2905         connp->conn_trace_last = last;
2906         return (1);
2907 }
2908 #endif
```

```
**********************************************************
    6502 Mon Jul  9 14:38:18 2012
new/usr/src/uts/common/inet/ip_impl.h
dccp: reset packet
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _INET_IP_IMPL_H
#define _INET_IP_IMPL_H

/*
 * IP implementation private declarations.  These interfaces are
 * used to build the IP module and are not meant to be accessed
 * by any modules except IP itself.  They are undocumented and are
 * subject to change without notice.
 */

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef _KERNEL

#include <sys/sdt.h>
#include <sys/dld.h>
#include <inet/tunables.h>

#define IP_MOD_ID               5701

#define INET_NAME       "ip"

#ifdef  _BIG_ENDIAN
#define IP_HDR_CSUM_TTL_ADJUST  256
#define IP_TCP_CSUM_COMP        IPPROTO_TCP
#define IP_UDP_CSUM_COMP        IPPROTO_UDP
#define IP_ICMPV6_CSUM_COMP     IPPROTO_ICMPV6
#define IP_DCCP_CSUM_COMP       IPPROTO_DCCP
#endif /* ! codereview */
#else
#define IP_HDR_CSUM_TTL_ADJUST  1
#define IP_TCP_CSUM_COMP        (IPPROTO_TCP << 8)
#define IP_UDP_CSUM_COMP        (IPPROTO_UDP << 8)
#define IP_ICMPV6_CSUM_COMP     (IPPROTO_ICMPV6 << 8)
```



```
#define IP_DCCP_CSUM_COMP       (IPPROTO_DCCP << 8)
#endif /* ! codereview */
#endif

#define TCP_CHECKSUM_OFFSET     16
#define TCP_CHECKSUM_SIZE       2

#define UDP_CHECKSUM_OFFSET     6
#define UDP_CHECKSUM_SIZE       2

#define ICMPV6_CHECKSUM_OFFSET  2
#define ICMPV6_CHECKSUM_SIZE    2

#define DCCP_CHECKSUM_OFFSET    6
#define DCCP_CHECKSUM_SIZE      2

#endif /* ! codereview */
#define IPH_TCPH_CHECKSUMP(ipha, hlen)  \
        ((uint16_t *)(((uchar_t *)(ipha)) + ((hlen) + TCP_CHECKSUM_OFFSET)))

#define IPH_UDPH_CHECKSUMP(ipha, hlen)  \
        ((uint16_t *)(((uchar_t *)(ipha)) + ((hlen) + UDP_CHECKSUM_OFFSET)))

#define IPH_ICMPV6_CHECKSUMP(ipha, hlen)        \
        ((uint16_t *)(((uchar_t *)(ipha)) + ((hlen) + ICMPV6_CHECKSUM_OFFSET)))

#define IPH_DCCPH_CHECKSUMP(ipha, hlen) \
        ((uint16_t *)(((uchar_t *)(ipha)) + ((hlen) + DCCP_CHECKSUM_OFFSET)))

#endif /* ! codereview */
#define ILL_HCKSUM_CAPABLE(ill)         \
        (((ill)->ill_capabilities & ILL_CAPAB_HCKSUM) != 0)

/*
 * Macro to adjust a given checksum value depending on any prepended
 * or postpended data on the packet.  It expects the start offset to
 * begin at an even boundary and that the packet consists of at most
 * two mblks.
 */
#define IP_ADJCKSUM_PARTIAL(cksum_start, mp, mp1, len, adj) {           \
        /*                                                              \
         * Prepended extraneous data; adjust checksum.                  \
         */                                                             \
        if ((len) > 0)                                                  \
                (adj) = IP_BCSUM_PARTIAL(cksum_start, len, 0);          \
        else                                                            \
                (adj) = 0;                                              \
        /*                                                              \
         * len is now the total length of mblk(s)                       \
         */                                                             \
        (len) = MBLKL(mp);                                              \
        if ((mp1) == NULL)                                              \
                (mp1) = (mp);                                           \
        else                                                            \
                (len) += MBLKL(mp1);                                    \
        /*                                                              \
         * Postpended extraneous data; adjust checksum.                 \
         */                                                             \
        if (((len) = (DB_CKSUMEND(mp) - len)) > 0) {                    \
                uint32_t _pad;                                          \
                                                                        \
                _pad = IP_BCSUM_PARTIAL((mp1)->b_wptr, len, 0);         \
                /*                                                      \
                 * If the postpended extraneous data was odd            \
                 * byte aligned, swap resulting checksum bytes.         \
                 */                                                     \
```

```
128                 if ((uintptr_t)(mp1)->b_wptr & 1)                       \
129                         (adj) += ((_pad << 8) & 0xFFFF) | (_pad >> 8);  \
130                 else                                                    \
131                         (adj) += _pad;                                  \
132                 (adj) = ((adj) & 0xFFFF) + ((int)(adj) >> 16);          \
133         }                                                               \
134 }

136 #define IS_SIMPLE_IPH(ipha)                                             \
137         ((ipha)->ipha_version_and_hdr_length == IP_SIMPLE_HDR_VERSION)

139 /*
140  * Currently supported flags for LSO.
141  */
142 #define LSO_BASIC_TCP_IPV4      DLD_LSO_BASIC_TCP_IPV4
143 #define LSO_BASIC_TCP_IPV6      DLD_LSO_BASIC_TCP_IPV6

145 #define ILL_LSO_CAPABLE(ill)                                            \
146         (((ill)->ill_capabilities & ILL_CAPAB_LSO) != 0)

148 #define ILL_LSO_USABLE(ill)                                             \
149         (ILL_LSO_CAPABLE(ill) &&                                        \
150         ill->ill_lso_capab != NULL)

152 #define ILL_LSO_TCP_IPV4_USABLE(ill)                                    \
153         (ILL_LSO_USABLE(ill) &&                                         \
154         ill->ill_lso_capab->ill_lso_flags & LSO_BASIC_TCP_IPV4)

156 #define ILL_LSO_TCP_IPV6_USABLE(ill)                                    \
157         (ILL_LSO_USABLE(ill) &&                                         \
158         ill->ill_lso_capab->ill_lso_flags & LSO_BASIC_TCP_IPV6)

160 #define ILL_ZCOPY_CAPABLE(ill)                                          \
161         (((ill)->ill_capabilities & ILL_CAPAB_ZEROCOPY) != 0)

163 #define ILL_ZCOPY_USABLE(ill)                                           \
164         (ILL_ZCOPY_CAPABLE(ill) && (ill->ill_zerocopy_capab != NULL) && \
165         (ill->ill_zerocopy_capab->ill_zerocopy_flags != 0))


168 /* Macro that follows definitions of flags for mac_tx() (see mac_client.h) */
169 #define IP_DROP_ON_NO_DESC      0x01    /* Equivalent to MAC_DROP_ON_NO_DESC */

171 #define ILL_DIRECT_CAPABLE(ill)                                         \
172         (((ill)->ill_capabilities & ILL_CAPAB_DLD_DIRECT) != 0)

174 /* This macro is used by the mac layer */
175 #define MBLK_RX_FANOUT_SLOWPATH(mp, ipha)                               \
176         (DB_TYPE(mp) != M_DATA || DB_REF(mp) != 1 || !OK_32PTR(ipha) || \
177         (((uchar_t *)ipha + IP_SIMPLE_HDR_LENGTH) >= (mp)->b_wptr))

179 /*
180  * In non-global zone exclusive IP stacks, data structures such as IRE
181  * entries pretend that they're in the global zone.  The following
182  * macro evaluates to the real zoneid instead of a pretend
183  * GLOBAL_ZONEID.
184  */
185 #define IP_REAL_ZONEID(zoneid, ipst)                                    \
186         (((zoneid) == GLOBAL_ZONEID) ?                                  \
187             netstackid_to_zoneid((ipst)->ips_netstack->netstack_stackid) : \
188             (zoneid))

190 extern void ill_flow_enable(void *, ip_mac_tx_cookie_t);
191 extern zoneid_t ip_get_zoneid_v4(ipaddr_t, mblk_t *, ip_recv_attr_t *,
192     zoneid_t);
193 extern zoneid_t ip_get_zoneid_v6(in6_addr_t *, mblk_t *, const ill_t *,
```



```
194     ip_recv_attr_t *, zoneid_t);
195 extern void conn_ire_revalidate(conn_t *, void *);
196 extern void ip_ire_unbind_walker(ire_t *, void *);
197 extern void ip_ire_rebind_walker(ire_t *, void *);

199 /*
200  * flag passed in by IP based protocols to get a private ip stream with
201  * no conn_t. Note this flag has the same value as SO_FALLBACK
202  */
203 #define IP_HELPER_STR   SO_FALLBACK

205 #define IP_MOD_MINPSZ   1
206 #define IP_MOD_MAXPSZ   INFPSZ
207 #define IP_MOD_HIWAT    65536
208 #define IP_MOD_LOWAT    1024

210 #define DEV_IP  "/devices/pseudo/ip@0:ip"
211 #define DEV_IP6 "/devices/pseudo/ip6@0:ip6"

213 #endif  /* _KERNEL */

215 #ifdef  __cplusplus
216 }
217 #endif

219 #endif  /* _INET_IP_IMPL_H */
```

```
**********************************************************
   13690 Mon Jul  9 14:38:18 2012
new/usr/src/uts/common/inet/ip_stack.h
dccp: ips_ipcl_dccp_fanout
**********************************************************
______unchanged_portion_omitted_

 143 /*
 144  * IP stack instances
 145  */
 146 struct ip_stack {
 147         netstack_t      *ips_netstack;  /* Common netstack */

 149         uint_t                  ips_src_generation;     /* Both IPv4 and IPv6 */

 151         struct mod_prop_info_s  *ips_propinfo_tbl;      /* ip tunables table */

 153         mib2_ipIfStatsEntry_t   ips_ip_mib;     /* SNMP fixed size info */
 154         mib2_icmp_t     ips_icmp_mib;
 155         /*
 156          * IPv6 mibs when the interface (ill) is not known.
 157          * When the ill is known the per-interface mib in the ill is used.
 158          */
 159         mib2_ipIfStatsEntry_t   ips_ip6_mib;
 160         mib2_ipv6IfIcmpEntry_t  ips_icmp6_mib;

 162         struct igmpstat         ips_igmpstat;

 164         kstat_t         *ips_ip_mibkp;  /* kstat exporting ip_mib data */
 165         kstat_t         *ips_icmp_mibkp; /* kstat exporting icmp_mib data */
 166         kstat_t         *ips_ip_kstat;
 167         ip_stat_t       ips_ip_statistics;
 168         kstat_t         *ips_ip6_kstat;
 169         ip6_stat_t      ips_ip6_statistics;

 171 /* ip.c */
 172         kmutex_t        ips_igmp_timer_lock;
 173         kmutex_t        ips_mld_timer_lock;
 174         kmutex_t        ips_ip_mi_lock;
 175         kmutex_t        ips_ip_addr_avail_lock;
 176         krwlock_t       ips_ill_g_lock;

 178         krwlock_t       ips_ill_g_usesrc_lock;

 180         /* Taskq dispatcher for capability operations */
 181         kmutex_t        ips_capab_taskq_lock;
 182         kcondvar_t      ips_capab_taskq_cv;
 183         mblk_t          *ips_capab_taskq_head;
 184         mblk_t          *ips_capab_taskq_tail;
 185         kthread_t       *ips_capab_taskq_thread;
 186         boolean_t       ips_capab_taskq_quit;

 188 /* ipclassifier.c - keep in ip_stack_t */
 189         /* ipclassifier hash tables */
 190         struct connf_s  *ips_rts_clients;
 191         struct connf_s  *ips_ipcl_conn_fanout;
 192         struct connf_s  *ips_ipcl_bind_fanout;
 193         struct connf_s  *ips_ipcl_proto_fanout_v4;
 194         struct connf_s  *ips_ipcl_proto_fanout_v6;
 195         struct connf_s  *ips_ipcl_udp_fanout;
 196         struct connf_s  *ips_ipcl_raw_fanout;           /* RAW SCTP sockets */
 197         struct connf_s  *ips_ipcl_iptun_fanout;
 198         struct connf_s  *ips_ipcl_dccp_fanout;
 199 #endif /* ! codereview */
 200         uint_t          ips_ipcl_conn_fanout_size;
 201         uint_t          ips_ipcl_bind_fanout_size;
```



```
 202         uint_t          ips_ipcl_udp_fanout_size;
 203         uint_t          ips_ipcl_raw_fanout_size;
 204         uint_t          ips_ipcl_iptun_fanout_size;
 205         uint_t          ips_ipcl_dccp_fanout_size;
 206 #endif /* ! codereview */
 207         struct connf_s  *ips_ipcl_globalhash_fanout;
 208         int             ips_conn_g_index;

 210 /* ip.c */
 211         /* Following protected by igmp_timer_lock */
 212         int             ips_igmp_time_to_next;  /* Time since last timeout */
 213         int             ips_igmp_timer_scheduled_last;
 214         int             ips_igmp_deferred_next;
 215         timeout_id_t    ips_igmp_timeout_id;
 216         boolean_t       ips_igmp_timer_setter_active;

 218         /* Following protected by mld_timer_lock */
 219         int             ips_mld_time_to_next;   /* Time since last timeout */
 220         int             ips_mld_timer_scheduled_last;
 221         int             ips_mld_deferred_next;
 222         timeout_id_t    ips_mld_timeout_id;
 223         boolean_t       ips_mld_timer_setter_active;

 225         /* Protected by igmp_slowtimeout_lock */
 226         timeout_id_t    ips_igmp_slowtimeout_id;
 227         kmutex_t        ips_igmp_slowtimeout_lock;

 229         /* Protected by mld_slowtimeout_lock */
 230         timeout_id_t    ips_mld_slowtimeout_id;
 231         kmutex_t        ips_mld_slowtimeout_lock;

 233         /* IPv4 forwarding table */
 234         struct radix_node_head *ips_ip_ftable;

 236 #define IPV6_ABITS              128
 237 #define IP6_MASK_TABLE_SIZE     (IPV6_ABITS + 1)        /* 129 ptrs */
 238         struct irb      *ips_ip_forwarding_table_v6[IP6_MASK_TABLE_SIZE];

 240         /*
 241          * ire_ft_init_lock is used while initializing ip_forwarding_table
 242          * dynamically in ire_add.
 243          */
 244         kmutex_t        ips_ire_ft_init_lock;

 246         /*
 247          * This is the IPv6 counterpart of RADIX_NODE_HEAD_LOCK. It is used
 248          * to prevent adds and deletes while we are doing a ftable_lookup
 249          * and extracting the ire_generation.
 250          */
 251         krwlock_t       ips_ip6_ire_head_lock;

 253         uint32_t        ips_ip6_ftable_hash_size;

 255         ire_stats_t     ips_ire_stats_v4;       /* IPv4 ire statistics */
 256         ire_stats_t     ips_ire_stats_v6;       /* IPv6 ire statistics */

 258         /* Count how many condemned objects for kmem_cache callbacks */
 259         uint32_t        ips_num_ire_condemned;
 260         uint32_t        ips_num_nce_condemned;
 261         uint32_t        ips_num_dce_condemned;

 263         struct ire_s    *ips_ire_reject_v4;     /* For unreachable dests */
 264         struct ire_s    *ips_ire_reject_v6;     /* For unreachable dests */
 265         struct ire_s    *ips_ire_blackhole_v4;  /* For temporary failures */
 266         struct ire_s    *ips_ire_blackhole_v6;  /* For temporary failures */
```

```
268         /* ips_ire_dep_lock protects ire_dep_* relationship between IREs */
269         krwlock_t       ips_ire_dep_lock;

271         /* Destination Cache Entries */
272         struct dce_s    *ips_dce_default;
273         uint_t          ips_dce_hashsize;
274         struct dcb_s    *ips_dce_hash_v4;
275         struct dcb_s    *ips_dce_hash_v6;

277         /* pending binds */
278         mblk_t          *ips_ip6_asp_pending_ops;
279         mblk_t          *ips_ip6_asp_pending_ops_tail;

281         /* Synchronize updates with table usage */
282         mblk_t          *ips_ip6_asp_pending_update; /* pending table updates */

284         boolean_t       ips_ip6_asp_uip;        /* table update in progress */
285         kmutex_t        ips_ip6_asp_lock;       /* protect all the above */
286         uint32_t        ips_ip6_asp_refcnt;     /* outstanding references */

288         struct ip6_asp  *ips_ip6_asp_table;
289         /* The number of policy entries in the table */
290         uint_t          ips_ip6_asp_table_count;

292         struct conn_s   *ips_ip_g_mrouter;

294         /* Time since last icmp_pkt_err */
295         clock_t         ips_icmp_pkt_err_last;
296         /* Number of packets sent in burst */
297         uint_t          ips_icmp_pkt_err_sent;

299         /* Protected by ip_mi_lock */
300         void            *ips_ip_g_head; /* IP Instance Data List Head */
301         void            *ips_arp_g_head; /* ARP Instance Data List Head */

303         /* Multirouting stuff */
304         /* Interval (in ms) between consecutive 'bad MTU' warnings */
305         hrtime_t        ips_ip_multirt_log_interval;
306         /* Time since last warning issued. */
307         hrtime_t        ips_multirt_bad_mtu_last_time;

309         /*
310          * CGTP hooks. Enabling and disabling of hooks is controlled by an
311          * IP tunable 'ips_ip_cgtp_filter'.
312          */
313         struct cgtp_filter_ops *ips_ip_cgtp_filter_ops;

315         struct ipsq_s   *ips_ipsq_g_head;
316         uint_t          ips_ill_index;  /* Used to assign interface indicies */
317         /* When set search for unused index */
318         boolean_t       ips_ill_index_wrap;

320         uint_t          ips_loopback_packets;

322         /* NDP/NCE structures for IPv4 and IPv6 */
323         struct ndp_g_s  *ips_ndp4;
324         struct ndp_g_s  *ips_ndp6;

326         /* ip_mroute stuff */
327         kmutex_t        ips_ip_g_mrouter_mutex;

329         struct mrtstat  *ips_mrtstat;   /* Stats for netstat */
330         int             ips_saved_ip_forwarding;

332         /* numvifs is only a hint about the max interface being used. */
333         ushort_t        ips_numvifs;
```



```
334         kmutex_t        ips_numvifs_mutex;

336         struct vif      *ips_vifs;
337         struct mfcb     *ips_mfcs;      /* kernel routing table */
338         struct tbf      *ips_tbfs;
339         /*
340          * One-back cache used to locate a tunnel's vif,
341          * given a datagram's src ip address.
342          */
343         ipaddr_t        ips_last_encap_src;
344         struct vif      *ips_last_encap_vif;
345         kmutex_t        ips_last_encap_lock;    /* Protects the above */

347         /*
348          * reg_vif_num is protected by numvifs_mutex
349          */
350         /* Whether or not special PIM assert processing is enabled. */
351         ushort_t        ips_reg_vif_num;        /* Index to Register vif */
352         int             ips_pim_assert;

354         union ill_g_head_u *ips_ill_g_heads;   /* ILL List Head */

356         kstat_t         *ips_loopback_ksp;

358         /* Array of conn drain lists */
359         struct idl_tx_list_s    *ips_idl_tx_list;
360         uint_t          ips_conn_drain_list_cnt; /* Count of conn_drain_list */

362         /*
363          * ID used to assign next free one.
364          * Increases by one. Once it wraps we search for an unused ID.
365          */
366         uint_t          ips_ip_src_id;
367         boolean_t       ips_srcid_wrapped;

369         struct srcid_map *ips_srcid_head;
370         krwlock_t       ips_srcid_lock;

372         uint64_t        ips_ipif_g_seqid;       /* Used only for sctp_addr.c */
373         union phyint_list_u *ips_phyint_g_list; /* start of phyint list */

375 /* ip_netinfo.c */
376         hook_family_t   ips_ipv4root;
377         hook_family_t   ips_ipv6root;
378         hook_family_t   ips_arproot;

380         net_handle_t            ips_ipv4_net_data;
381         net_handle_t            ips_ipv6_net_data;
382         net_handle_t            ips_arp_net_data;

384         /*
385          * Hooks for firewalling
386          */
387         hook_event_t            ips_ip4_physical_in_event;
388         hook_event_t            ips_ip4_physical_out_event;
389         hook_event_t            ips_ip4_forwarding_event;
390         hook_event_t            ips_ip4_loopback_in_event;
391         hook_event_t            ips_ip4_loopback_out_event;

393         hook_event_t            ips_ip6_physical_in_event;
394         hook_event_t            ips_ip6_physical_out_event;
395         hook_event_t            ips_ip6_forwarding_event;
396         hook_event_t            ips_ip6_loopback_in_event;
397         hook_event_t            ips_ip6_loopback_out_event;

399         hook_event_t            ips_arp_physical_in_event;
```

```
400         hook_event_t            ips_arp_physical_out_event;
401         hook_event_t            ips_arp_nic_events;

403         hook_event_token_t      ips_ipv4firewall_physical_in;
404         hook_event_token_t      ips_ipv4firewall_physical_out;
405         hook_event_token_t      ips_ipv4firewall_forwarding;
406         hook_event_token_t      ips_ipv4firewall_loopback_in;
407         hook_event_token_t      ips_ipv4firewall_loopback_out;

409         hook_event_token_t      ips_ipv6firewall_physical_in;
410         hook_event_token_t      ips_ipv6firewall_physical_out;
411         hook_event_token_t      ips_ipv6firewall_forwarding;
412         hook_event_token_t      ips_ipv6firewall_loopback_in;
413         hook_event_token_t      ips_ipv6firewall_loopback_out;

415         hook_event_t            ips_ip4_nic_events;
416         hook_event_t            ips_ip6_nic_events;
417         hook_event_token_t      ips_ipv4nicevents;
418         hook_event_token_t      ips_ipv6nicevents;

420         hook_event_token_t      ips_arp_physical_in;
421         hook_event_token_t      ips_arp_physical_out;
422         hook_event_token_t      ips_arpnicevents;

424         net_handle_t            ips_ip4_observe_pr;
425         net_handle_t            ips_ip6_observe_pr;
426         hook_event_t            ips_ip4_observe;
427         hook_event_t            ips_ip6_observe;
428         hook_event_token_t      ips_ipv4observing;
429         hook_event_token_t      ips_ipv6observing;

431         struct __ldi_ident      *ips_ldi_ident;

433 /* ipmp.c */
434         krwlock_t               ips_ipmp_lock;
435         mod_hash_t              *ips_ipmp_grp_hash;

437 };
438 typedef struct ip_stack ip_stack_t;

440 /* Finding an ip_stack_t */
441 #define CONNQ_TO_IPST(_q)       (Q_TO_CONN(_q)->conn_netstack->netstack_ip)
442 #define ILLQ_TO_IPST(_q)        (((ill_t *)(_q)->q_ptr)->ill_ipst)
443 #define PHYINT_TO_IPST(phyi)    ((phyi)->phyint_ipsq->ipsq_ipst)

445 #else /* _KERNEL */
446 typedef int ip_stack_t;
447 #endif /* _KERNEL */

449 #ifdef  __cplusplus
450 }
451 #endif

453 #endif  /* _INET_IP_STACK_H */
```

```
**********************************************************
   26743 Mon Jul  9 14:38:18 2012
new/usr/src/uts/common/inet/ipclassifier.h
dccp: conn_t
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _INET_IPCLASSIFIER_H
#define _INET_IPCLASSIFIER_H

#ifdef __cplusplus
extern "C" {
#endif

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/mi.h>
#include <inet/tcp.h>
#include <inet/ip6.h>
#include <netinet/in.h>         /* for IPPROTO_* constants */
#include <sys/sdt.h>
#include <sys/socket_proto.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>

typedef void (*edesc_rpf)(void *, mblk_t *, void *, ip_recv_attr_t *);
struct icmph_s;
struct icmp6_hdr;
typedef boolean_t (*edesc_vpf)(conn_t *, void *, struct icmph_s *,
    struct icmp6_hdr *, ip_recv_attr_t *);

/*
 * ==============================
 * =    The CONNECTION          =
 * ==============================
 */

/*
 * The connection structure contains the common information/flags/ref needed.
 * Implementation will keep the connection struct, the layers (with their
 * respective data for event i.e. tcp_t if event was tcp_input_data) all in one
 * contiguous memory location.
 */
```



```
/* Conn Flags */
/* Unused                       0x00020000 */
/* Unused                       0x00040000 */
#define IPCL_FULLY_BOUND        0x00080000      /* Bound to correct squeue */
/* Unused                       0x00100000 */
/* Unused                       0x00200000 */
/* Unused                       0x00400000 */
#define IPCL_CL_LISTENER        0x00800000      /* Cluster listener */
/* Unused                       0x01000000 */
/* Unused                       0x02000000 */
/* Unused                       0x04000000 */
/* Unused                       0x08000000 */
/* Unused                       0x10000000 */
/* Unused                       0x20000000 */
#define IPCL_CONNECTED          0x40000000      /* Conn in connected table */
#define IPCL_BOUND              0x80000000      /* Conn in bind table */

/* Flags identifying the type of conn */
#define IPCL_TCPCONN            0x00000001      /* From tcp_conn_cache */
#define IPCL_SCTPCONN           0x00000002      /* From sctp_conn_cache */
#define IPCL_IPCCONN            0x00000004      /* From ip_conn_cache */
#define IPCL_UDPCONN            0x00000008      /* From udp_conn_cache */
#define IPCL_RAWIPCONN          0x00000010      /* From rawip_conn_cache */
#define IPCL_RTSCONN            0x00000020      /* From rts_conn_cache */
#define IPCL_DCCPCONN           0x00000040      /* From dccp_conn_cache */
/* Unused                       0x00000040 */
#define IPCL_IPTUN              0x00000080      /* iptun module above us */

#define IPCL_NONSTR             0x00001000      /* A non-STREAMS socket */
/* Unused                       0x10000000 */

#define IPCL_REMOVED            0x00000100
#define IPCL_REUSED             0x00000200

#define IPCL_IS_CONNECTED(connp)                                        \
        ((connp)->conn_flags & IPCL_CONNECTED)

#define IPCL_IS_BOUND(connp)                                            \
        ((connp)->conn_flags & IPCL_BOUND)

/*
 * Can't use conn_proto since we need to tell difference
 * between a real TCP socket and a SOCK_RAW, IPPROTO_TCP.
 */
#define IPCL_IS_TCP(connp)                                              \
        ((connp)->conn_flags & IPCL_TCPCONN)

#define IPCL_IS_SCTP(connp)                                             \
        ((connp)->conn_flags & IPCL_SCTPCONN)

#define IPCL_IS_UDP(connp)                                              \
        ((connp)->conn_flags & IPCL_UDPCONN)

#define IPCL_IS_RAWIP(connp)                                            \
        ((connp)->conn_flags & IPCL_RAWIPCONN)

#define IPCL_IS_RTS(connp)                                              \
        ((connp)->conn_flags & IPCL_RTSCONN)

#define IPCL_IS_IPTUN(connp)                                            \
        ((connp)->conn_flags & IPCL_IPTUN)

#define IPCL_IS_DCCP(connp)                                             \
        ((connp)->conn_flags & IPCL_DCCPCONN)
```

```
#endif /* ! codereview */
#define IPCL_IS_NONSTR(connp)   ((connp)->conn_flags & IPCL_NONSTR)

typedef struct connf_s connf_t;

typedef struct
{
        int     ctb_depth;
#define CONN_STACK_DEPTH        15
        pc_t    ctb_stack[CONN_STACK_DEPTH];
} conn_trace_t;

typedef struct ip_helper_minor_info_s {
        dev_t   ip_minfo_dev;           /* Device */
        vmem_t  *ip_minfo_arena;        /* Arena */
} ip_helper_minfo_t;

/*
 * ip helper stream info
 */
typedef struct ip_helper_stream_info_s {
        ldi_handle_t            iphs_handle;
        queue_t                 *iphs_rq;
        queue_t                 *iphs_wq;
        ip_helper_minfo_t       *iphs_minfo;
} ip_helper_stream_info_t;

/*
 * Mandatory Access Control mode, in conn_t's conn_mac_mode field.
 *      CONN_MAC_DEFAULT: strict enforcement of MAC.
 *      CONN_MAC_AWARE:   allows communications between unlabeled systems
 *                        and privileged daemons
 *      CONN_MAC_IMPLICIT: allows communications without explicit labels
 *                         on the wire with privileged daemons.
 *
 * CONN_MAC_IMPLICIT is intended specifically for labeled IPsec key management
 * in networks which don't pass CIPSO-labeled packets.
 */
#define CONN_MAC_DEFAULT 0
#define CONN_MAC_AWARE 1
#define CONN_MAC_IMPLICIT 2

/*
 * conn receive ancillary definition.
 *
 * These are the set of socket options that make the receive side
 * potentially pass up ancillary data items.
 * We have a union with an integer so that we can quickly check whether
 * any ancillary data items need to be added.
 */
typedef struct crb_s {
        union {
                uint32_t        crbu_all;
                struct {
                        uint32_t
        crbb_recvdstaddr : 1,           /* IP_RECVDSTADDR option */
        crbb_recvopts : 1,              /* IP_RECVOPTS option */
        crbb_recvif : 1,                /* IP_RECVIF option */
        crbb_recvslla : 1,              /* IP_RECVSLLA option */

        crbb_recvttl : 1,               /* IP_RECVTTL option */
        crbb_ip_recvpktinfo : 1,        /* IP*_RECVPKTINFO option  */
        crbb_ipv6_recvhoplimit : 1,     /* IPV6_RECVHOPLIMIT option */
        crbb_ipv6_recvhopopts : 1,      /* IPV6_RECVHOPOPTS option */

        crbb_ipv6_recvdstopts : 1,      /* IPV6_RECVDSTOPTS option */
```



```
        crbb_ipv6_recvrthdr : 1,        /* IPV6_RECVRTHDR option */
        crbb_old_ipv6_recvdstopts : 1,  /* old form of IPV6_DSTOPTS */
        crbb_ipv6_recvrthdrdstopts : 1, /* IPV6_RECVRTHDRDSTOPTS */

        crbb_ipv6_recvtclass : 1,       /* IPV6_RECVTCLASS */
        crbb_recvucred : 1,             /* IP_RECVUCRED option */
        crbb_timestamp : 1;             /* SO_TIMESTAMP "socket" option */

                } crbb;
        } crbu;
} crb_t;

#define crb_all                         crbu.crbu_all
#define crb_recvdstaddr                 crbu.crbb.crbb_recvdstaddr
#define crb_recvopts                    crbu.crbb.crbb_recvopts
#define crb_recvif                      crbu.crbb.crbb_recvif
#define crb_recvslla                    crbu.crbb.crbb_recvslla
#define crb_recvttl                     crbu.crbb.crbb_recvttl
#define crb_ip_recvpktinfo              crbu.crbb.crbb_ip_recvpktinfo
#define crb_ipv6_recvhoplimit           crbu.crbb.crbb_ipv6_recvhoplimit
#define crb_ipv6_recvhopopts            crbu.crbb.crbb_ipv6_recvhopopts
#define crb_ipv6_recvdstopts            crbu.crbb.crbb_ipv6_recvdstopts
#define crb_ipv6_recvrthdr              crbu.crbb.crbb_ipv6_recvrthdr
#define crb_old_ipv6_recvdstopts        crbu.crbb.crbb_old_ipv6_recvdstopts
#define crb_ipv6_recvrthdrdstopts       crbu.crbb.crbb_ipv6_recvrthdrdstopts
#define crb_ipv6_recvtclass             crbu.crbb.crbb_ipv6_recvtclass
#define crb_recvucred                   crbu.crbb.crbb_recvucred
#define crb_timestamp                   crbu.crbb.crbb_timestamp

/*
 * The initial fields in the conn_t are setup by the kmem_cache constructor,
 * and are preserved when it is freed. Fields after that are bzero'ed when
 * the conn_t is freed.
 *
 * Much of the conn_t is protected by conn_lock.
 *
 * conn_lock is also used by some ULPs (like UDP and RAWIP) to protect
 * their state.
 */
struct conn_s {
        kmutex_t        conn_lock;
        uint32_t        conn_ref;               /* Reference counter */
        uint32_t        conn_flags;             /* Conn Flags */

        union {
                tcp_t           *cp_tcp;        /* Pointer to the tcp struct */
                struct udp_s    *cp_udp;        /* Pointer to the udp struct */
                struct icmp_s   *cp_icmp;       /* Pointer to rawip struct */
                struct rts_s    *cp_rts;        /* Pointer to rts struct */
                struct iptun_s  *cp_iptun;      /* Pointer to iptun_t */
                struct sctp_s   *cp_sctp;       /* For IPCL_SCTPCONN */
                struct dccp_s   *cp_dccp;       /* Pointer to dccp struct */
#endif /* ! codereview */
                void            *cp_priv;
        } conn_proto_priv;
#define conn_tcp        conn_proto_priv.cp_tcp
#define conn_udp        conn_proto_priv.cp_udp
#define conn_icmp       conn_proto_priv.cp_icmp
#define conn_rts        conn_proto_priv.cp_rts
#define conn_iptun      conn_proto_priv.cp_iptun
#define conn_sctp       conn_proto_priv.cp_sctp
#define conn_dccp       conn_proto_priv.cp_dccp
#endif /* ! codereview */
#define conn_priv       conn_proto_priv.cp_priv

        kcondvar_t      conn_cv;
```

```
259         uint8_t         conn_proto;             /* protocol type */

261         edesc_rpf       conn_recv;              /* Pointer to recv routine */
262         edesc_rpf       conn_recvicmp;          /* For ICMP error */
263         edesc_vpf       conn_verifyicmp;        /* Verify ICMP error */

265         ip_xmit_attr_t  *conn_ixa;              /* Options if no ancil data */

267         /* Fields after this are bzero'ed when the conn_t is freed. */
268 #define conn_start_clr  conn_recv_ancillary

270         /* Options for receive-side ancillary data */
271         crb_t           conn_recv_ancillary;

273         squeue_t        *conn_sqp;              /* Squeue for processing */
274         uint_t          conn_state_flags;       /* IP state flags */

276         int             conn_lingertime;        /* linger time (in seconds) */

278         unsigned int
279                 conn_on_sqp : 1,                /* Conn is being processed */
280                 conn_linger : 1,                /* SO_LINGER state */
281                 conn_useloopback : 1,           /* SO_USELOOPBACK state */
282                 conn_broadcast : 1,             /* SO_BROADCAST state */

284                 conn_reuseaddr : 1,             /* SO_REUSEADDR state */
285                 conn_keepalive : 1,             /* SO_KEEPALIVE state */
286                 conn_multi_router : 1,          /* Wants all multicast pkts */
287                 conn_unspec_src : 1,            /* IP_UNSPEC_SRC */

289                 conn_policy_cached : 1,         /* Is policy cached/latched ? */
290                 conn_in_enforce_policy : 1,     /* Enforce Policy on inbound */
291                 conn_out_enforce_policy : 1,    /* Enforce Policy on outbound */
292                 conn_debug : 1,                 /* SO_DEBUG */

294                 conn_ipv6_v6only : 1,           /* IPV6_V6ONLY */
295                 conn_oobinline : 1,             /* SO_OOBINLINE state */
296                 conn_dgram_errind : 1,          /* SO_DGRAM_ERRIND state */
297                 conn_exclbind : 1,              /* SO_EXCLBIND state */

299                 conn_mdt_ok : 1,                /* MDT is permitted */
300                 conn_allzones : 1,              /* SO_ALLZONES */
301                 conn_ipv6_recvpathmtu : 1,      /* IPV6_RECVPATHMTU */
302                 conn_mcbc_bind : 1,             /* Bound to multi/broadcast */

304                 conn_pad_to_bit_31 : 12;

306         boolean_t       conn_blocked;           /* conn is flow-controlled */

308         squeue_t        *conn_initial_sqp;      /* Squeue at open time */
309         squeue_t        *conn_final_sqp;        /* Squeue after connect */
310         ill_t           *conn_dhcpinit_ill;     /* IP_DHCPINIT_IF */
311         ipsec_latch_t   *conn_latch;            /* latched IDS */
312         struct ipsec_policy_s   *conn_latch_in_policy; /* latched policy (in) */
313         struct ipsec_action_s   *conn_latch_in_action; /* latched action (in) */
314         uint_t          conn_bound_if;          /* IP*_BOUND_IF */
315         queue_t         *conn_rq;               /* Read queue */
316         queue_t         *conn_wq;               /* Write queue */
317         dev_t           conn_dev;               /* Minor number */
318         vmem_t          *conn_minor_arena;      /* Minor arena */
319         ip_helper_stream_info_t *conn_helper_info;

321         cred_t          *conn_cred;             /* Credentials */
322         pid_t           conn_cpid;              /* pid from open/connect */
323         uint64_t        conn_open_time;         /* time when this was opened */
```



```
325         connf_t         *conn_g_fanout;         /* Global Hash bucket head */
326         struct conn_s   *conn_g_next;           /* Global Hash chain next */
327         struct conn_s   *conn_g_prev;           /* Global Hash chain prev */
328         struct ipsec_policy_head_s *conn_policy; /* Configured policy */
329         in6_addr_t      conn_bound_addr_v6;     /* Address in bind() */
330 #define conn_bound_addr_v4      V4_PART_OF_V6(conn_bound_addr_v6)
331         connf_t         *conn_fanout;           /* Hash bucket we're part of */
332         struct conn_s   *conn_next;             /* Hash chain next */
333         struct conn_s   *conn_prev;             /* Hash chain prev */

335         struct {
336                 in6_addr_t connua_laddr;        /* Local address - match */
337                 in6_addr_t connua_faddr;        /* Remote address */
338         } connua_v6addr;
339 #define conn_laddr_v4   V4_PART_OF_V6(connua_v6addr.connua_laddr)
340 #define conn_faddr_v4   V4_PART_OF_V6(connua_v6addr.connua_faddr)
341 #define conn_laddr_v6   connua_v6addr.connua_laddr
342 #define conn_faddr_v6   connua_v6addr.connua_faddr
343         in6_addr_t      conn_saddr_v6;          /* Local address - source */
344 #define conn_saddr_v4   V4_PART_OF_V6(conn_saddr_v6)

346         union {
347                 /* Used for classifier match performance */
348                 uint32_t                connu_ports2;
349                 struct {
350                         in_port_t       connu_fport;    /* Remote port */
351                         in_port_t       connu_lport;    /* Local port */
352                 } connu_ports;
353         } u_port;
354 #define conn_fport      u_port.connu_ports.connu_fport
355 #define conn_lport      u_port.connu_ports.connu_lport
356 #define conn_ports      u_port.connu_ports2

358         uint_t          conn_incoming_ifindex;  /* IP{,V6}_BOUND_IF, scopeid */
359         ill_t           *conn_oper_pending_ill; /* pending shared ioctl */

361         krwlock_t       conn_ilg_lock;          /* Protects conn_ilg_* */
362         ilg_t           *conn_ilg;              /* Group memberships */

364         kcondvar_t      conn_refcv;             /* For conn_oper_pending_ill */

366         struct conn_s   *conn_drain_next;       /* Next conn in drain list */
367         struct conn_s   *conn_drain_prev;       /* Prev conn in drain list */
368         idl_t           *conn_idl;              /* Ptr to the drain list head */
369         mblk_t          *conn_ipsec_opt_mp;     /* ipsec option mblk */
370         zoneid_t        conn_zoneid;            /* zone connection is in */
371         int             conn_rtaware;           /* RT_AWARE sockopt value */
372         kcondvar_t      conn_sq_cv;             /* For non-STREAMS socket IO */
373         sock_upcalls_t  *conn_upcalls;          /* Upcalls to sockfs */
374         sock_upper_handle_t conn_upper_handle;  /* Upper handle: sonode * */

376         unsigned int
377                 conn_mlp_type : 2,              /* mlp_type_t; tsol/tndb.h */
378                 conn_anon_mlp : 1,              /* user wants anon MLP */
379                 conn_anon_port : 1,             /* user bound anonymously */

381                 conn_mac_mode : 2,              /* normal/loose/implicit MAC */
382                 conn_anon_priv_bind : 1,        /* *_ANON_PRIV_BIND state */
383                 conn_zone_is_global : 1,        /* GLOBAL_ZONEID */
384                 conn_isvrrp : 1,                /* VRRP control socket */
385                 conn_spare : 23;

387         boolean_t       conn_flow_cntrld;
388         netstack_t      *conn_netstack; /* Corresponds to a netstack_hold */

390         /*
```

```
391          * IP format that packets received for this struct should use.
392          * Value can be IP4_VERSION or IPV6_VERSION.
393          * The sending version is encoded using IXAF_IS_IPV4.
394          */
395         ushort_t        conn_ipversion;

397         /* Written to only once at the time of opening the endpoint */
398         sa_family_t     conn_family;            /* Family from socket() call */
399         uint_t          conn_so_type;           /* Type from socket() call */

401         uint_t          conn_sndbuf;            /* SO_SNDBUF state */
402         uint_t          conn_rcvbuf;            /* SO_RCVBUF state */
403         uint_t          conn_wroff;             /* Current write offset */

405         uint_t          conn_sndlowat;          /* Send buffer low water mark */
406         uint_t          conn_rcvlowat;          /* Recv buffer low water mark */

408         uint8_t         conn_default_ttl;       /* Default TTL/hoplimit */

410         uint32_t        conn_flowinfo;  /* Connected flow id and tclass */

412         /*
413          * The most recent address for sendto. Initially set to zero
414          * which is always different than then the destination address
415          * since the send interprets zero as the loopback address.
416          */
417         in6_addr_t      conn_v6lastdst;
418 #define conn_v4lastdst  V4_PART_OF_V6(conn_v6lastdst)
419         ushort_t        conn_lastipversion;
420         in_port_t       conn_lastdstport;
421         uint32_t        conn_lastflowinfo;      /* IPv6-only */
422         uint_t          conn_lastscopeid;       /* IPv6-only */
423         uint_t          conn_lastsrcid;         /* Only for AF_INET6 */
424         /*
425          * When we are not connected conn_saddr might be unspecified.
426          * We track the source that was used with conn_v6lastdst here.
427          */
428         in6_addr_t      conn_v6lastsrc;
429 #define conn_v4lastsrc  V4_PART_OF_V6(conn_v6lastsrc)

431         /* Templates for transmitting packets */
432         ip_pkt_t        conn_xmit_ipp;          /* Options if no ancil data */

434         /*
435          * Header template - conn_ht_ulp is a pointer into conn_ht_iphc.
436          * Note that ixa_ip_hdr_length indicates the offset of ht_ulp in
437          * ht_iphc
438          *
439          * The header template is maintained for connected endpoints (and
440          * updated when sticky options are changed) and also for the lastdst.
441          * There is no conflict between those usages since SOCK_DGRAM and
442          * SOCK_RAW can not be used to specify a destination address (with
443          * sendto/sendmsg) if the socket has been connected.
444          */
445         uint8_t         *conn_ht_iphc;          /* Start of IP header */
446         uint_t          conn_ht_iphc_allocated; /* Allocated buffer size */
447         uint_t          conn_ht_iphc_len;       /* IP+ULP size */
448         uint8_t         *conn_ht_ulp;           /* Upper-layer header */
449         uint_t          conn_ht_ulp_len;        /* ULP header len */

451         /* Checksum to compensate for source routed packets. Host byte order */
452         uint32_t        conn_sum;

454         uint32_t        conn_ioctlref;          /* ioctl ref count */
455 #ifdef CONN_DEBUG
456 #define CONN_TRACE_MAX  10
```



```
457         int             conn_trace_last;        /* ndx of last used tracebuf */
458         conn_trace_t    conn_trace_buf[CONN_TRACE_MAX];
459 #endif
460 };

462 /*
463  * connf_t - connection fanout data.
464  *
465  * The hash tables and their linkage (conn_t.{hashnextp, hashprevp} are
466  * protected by the per-bucket lock. Each conn_t inserted in the list
467  * points back at the connf_t that heads the bucket.
468  */
469 struct connf_s {
470         struct conn_s   *connf_head;
471         kmutex_t        connf_lock;
472 };

474 #define CONN_INC_REF(connp)     {                               \
475         mutex_enter(&(connp)->conn_lock);                       \
476         DTRACE_PROBE1(conn__inc__ref, conn_t *, connp);         \
477         ASSERT(conn_trace_ref(connp));                          \
478         (connp)->conn_ref++;                                    \
479         ASSERT((connp)->conn_ref != 0);                         \
480         mutex_exit(&(connp)->conn_lock);                        \
481 }

483 #define CONN_INC_REF_LOCKED(connp)      {                       \
484         DTRACE_PROBE1(conn__inc__ref, conn_t *, connp);         \
485         ASSERT(MUTEX_HELD(&(connp)->conn_lock));                \
486         ASSERT(conn_trace_ref(connp));                          \
487         (connp)->conn_ref++;                                    \
488         ASSERT((connp)->conn_ref != 0);                         \
489 }

491 #define CONN_DEC_REF(connp)     {                                       \
492         mutex_enter(&(connp)->conn_lock);                               \
493         DTRACE_PROBE1(conn__dec__ref, conn_t *, connp);                 \
494         /*                                                              \
495          * The squeue framework always does a CONN_DEC_REF after return \
496          * from TCP. Hence the refcnt must be at least 2 if conn_on_sqp \
497          * is B_TRUE and conn_ref is being decremented. This is to      \
498          * account for the mblk being currently processed.              \
499          */                                                             \
500         if ((connp)->conn_ref == 0 ||                                   \
501             ((connp)->conn_ref == 1 && (connp)->conn_on_sqp))           \
502                 cmn_err(CE_PANIC, "CONN_DEC_REF: connp(%p) has ref "    \
503                         "= %d\n", (void *)(connp), (connp)->conn_ref);  \
504         ASSERT(conn_untrace_ref(connp));                                \
505         (connp)->conn_ref--;                                            \
506         if ((connp)->conn_ref == 0) {                                   \
507                 /* Refcnt can't increase again, safe to drop lock */    \
508                 mutex_exit(&(connp)->conn_lock);                        \
509                 ipcl_conn_destroy(connp);                               \
510         } else {                                                        \
511                 cv_broadcast(&(connp)->conn_cv);                        \
512                 mutex_exit(&(connp)->conn_lock);                        \
513         }                                                               \
514 }

516 /*
517  * For use with subsystems within ip which use ALL_ZONES as a wildcard
518  */
519 #define IPCL_ZONEID(connp)                                              \
520         ((connp)->conn_allzones ? ALL_ZONES : (connp)->conn_zoneid)

522 /*
```

```
 * For matching between a conn_t and a zoneid.
 */
#define IPCL_ZONE_MATCH(connp, zoneid)                                  \
        (((connp)->conn_allzones) ||                                    \
            ((zoneid) == ALL_ZONES) ||                                  \
            (connp)->conn_zoneid == (zoneid))

/*
 * On a labeled system, we must treat bindings to ports
 * on shared IP addresses by sockets with MAC exemption
 * privilege as being in all zones, as there's
 * otherwise no way to identify the right receiver.
 */

#define IPCL_CONNS_MAC(conn1, conn2)                                    \
        (((conn1)->conn_mac_mode != CONN_MAC_DEFAULT) ||                \
        ((conn2)->conn_mac_mode != CONN_MAC_DEFAULT))

#define IPCL_BIND_ZONE_MATCH(conn1, conn2)                              \
        (IPCL_CONNS_MAC(conn1, conn2) ||                                \
        IPCL_ZONE_MATCH(conn1, conn2->conn_zoneid) ||                   \
        IPCL_ZONE_MATCH(conn2, conn1->conn_zoneid))


#define _IPCL_V4_MATCH(v6addr, v4addr)  \
        (V4_PART_OF_V6((v6addr)) == (v4addr) && IN6_IS_ADDR_V4MAPPED(&(v6addr)))

#define _IPCL_V4_MATCH_ANY(addr)        \
        (IN6_IS_ADDR_V4MAPPED_ANY(&(addr)) || IN6_IS_ADDR_UNSPECIFIED(&(addr)))


/*
 * IPCL_PROTO_MATCH() and IPCL_PROTO_MATCH_V6() only matches conns with
 * the specified ira_zoneid or conn_allzones by calling conn_wantpacket.
 */
#define IPCL_PROTO_MATCH(connp, ira, ipha)                              \
        ((((connp)->conn_laddr_v4 == INADDR_ANY) ||                     \
        (((connp)->conn_laddr_v4 == ((ipha)->ipha_dst)) &&              \
            (((connp)->conn_faddr_v4 == INADDR_ANY) ||                  \
        ((connp)->conn_faddr_v4 == ((ipha)->ipha_src))))) &&            \
        conn_wantpacket((connp), (ira), (ipha)))

#define IPCL_PROTO_MATCH_V6(connp, ira, ip6h)                           \
        ((IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) ||           \
        (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &((ip6h)->ip6_dst)) &&   \
        (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) ||                  \
        IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &((ip6h)->ip6_src))))) && \
        (conn_wantpacket_v6((connp), (ira), (ip6h))))

#define IPCL_CONN_HASH(src, ports, ipst)                                \
        ((unsigned)(ntohl((src)) ^ ((ports) >> 24) ^ ((ports) >> 16) ^  \
        ((ports) >> 8) ^ (ports)) % (ipst)->ips_ipcl_conn_fanout_size)

#define IPCL_CONN_HASH_V6(src, ports, ipst)                             \
        IPCL_CONN_HASH(V4_PART_OF_V6((src)), (ports), (ipst))

#define IPCL_CONN_MATCH(connp, proto, src, dst, ports)                  \
        ((connp)->conn_proto == (proto) &&                              \
                (connp)->conn_ports == (ports) &&                       \
                _IPCL_V4_MATCH((connp)->conn_faddr_v6, (src)) &&        \
                _IPCL_V4_MATCH((connp)->conn_laddr_v6, (dst)) &&        \
                !(connp)->conn_ipv6_v6only)

#define IPCL_CONN_MATCH_V6(connp, proto, src, dst, ports)               \
        ((connp)->conn_proto == (proto) &&                              \
                (connp)->conn_ports == (ports) &&                       \
```



```
                IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(src)) &&  \
                IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(dst)))

#define IPCL_PORT_HASH(port, size) \
        ((((port) >> 8) ^ (port)) & ((size) - 1))

#define IPCL_BIND_HASH(lport, ipst)                                     \
        ((unsigned)(((lport) >> 8) ^ (lport)) % \
            (ipst)->ips_ipcl_bind_fanout_size)

#define IPCL_BIND_MATCH(connp, proto, laddr, lport)                     \
        ((connp)->conn_proto == (proto) &&                              \
                (connp)->conn_lport == (lport) &&                       \
                (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) ||          \
                _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))) &&     \
                !(connp)->conn_ipv6_v6only)

#define IPCL_BIND_MATCH_V6(connp, proto, laddr, lport)                  \
        ((connp)->conn_proto == (proto) &&                              \
                (connp)->conn_lport == (lport) &&                       \
                (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) || \
                IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6)))

/*
 * We compare conn_laddr since it captures both connected and a bind to
 * a multicast or broadcast address.
 * The caller needs to match the zoneid and also call conn_wantpacket
 * for multicast, broadcast, or when conn_incoming_ifindex is set.
 */
#define IPCL_UDP_MATCH(connp, lport, laddr, fport, faddr)               \
        (((connp)->conn_lport == (lport)) &&                            \
        ((_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) ||                 \
        (_IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) &&             \
        (_IPCL_V4_MATCH_ANY((connp)->conn_faddr_v6) ||                  \
        (_IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)) &&             \
        (connp)->conn_fport == (fport)))))) &&                          \
        !(connp)->conn_ipv6_v6only)

/*
 * We compare conn_laddr since it captures both connected and a bind to
 * a multicast or broadcast address.
 * The caller needs to match the zoneid and also call conn_wantpacket_v6
 * for multicast or when conn_incoming_ifindex is set.
 */
#define IPCL_UDP_MATCH_V6(connp, lport, laddr, fport, faddr)    \
        (((connp)->conn_lport == (lport)) &&                    \
        (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) ||    \
        (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) &&       \
        (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) ||    \
        (IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(faddr)) &&       \
        (connp)->conn_fport == (fport))))))

#define IPCL_IPTUN_HASH(laddr, faddr)                                   \
        ((ntohl(laddr) ^ ((ntohl(faddr) << 24) | (ntohl(faddr) >> 8))) % \
        ipcl_iptun_fanout_size)

#define IPCL_IPTUN_HASH_V6(laddr, faddr)                                \
        IPCL_IPTUN_HASH((laddr)->s6_addr32[0] ^ (laddr)->s6_addr32[1] ^ \
            (faddr)->s6_addr32[2] ^ (faddr)->s6_addr32[3],              \
            (faddr)->s6_addr32[0] ^ (faddr)->s6_addr32[1] ^             \
            (laddr)->s6_addr32[2] ^ (laddr)->s6_addr32[3])

#define IPCL_IPTUN_MATCH(connp, laddr, faddr)                   \
        (_IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) &&     \
        _IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)))
```

```
#define IPCL_IPTUN_MATCH_V6(connp, laddr, faddr)                \
        (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, (laddr)) &&        \
        IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, (faddr)))

#define IPCL_UDP_HASH(lport, ipst)      \
        IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_udp_fanout_size)

#define IPCL_DCCP_HASH(lport, ipst)     \
        IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_dccp_fanout_size)

#endif /* ! codereview */
#define CONN_G_HASH_SIZE        1024

/* Raw socket hash function. */
#define IPCL_RAW_HASH(lport, ipst)      \
        IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_raw_fanout_size)

/*
 * This is similar to IPCL_BIND_MATCH except that the local port check
 * is changed to a wildcard port check.
 * We compare conn_laddr since it captures both connected and a bind to
 * a multicast or broadcast address.
 */
#define IPCL_RAW_MATCH(connp, proto, laddr)                     \
        ((connp)->conn_proto == (proto) &&                      \
        (connp)->conn_lport == 0 &&                             \
        (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) ||          \
        _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))))

#define IPCL_RAW_MATCH_V6(connp, proto, laddr)                  \
        ((connp)->conn_proto == (proto) &&                      \
        (connp)->conn_lport == 0 &&                             \
        (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) ||    \
        IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr))))

/* Function prototypes */
extern void ipcl_g_init(void);
extern void ipcl_init(ip_stack_t *);
extern void ipcl_g_destroy(void);
extern void ipcl_destroy(ip_stack_t *);
extern conn_t *ipcl_conn_create(uint32_t, int, netstack_t *);
extern void ipcl_conn_destroy(conn_t *);

void ipcl_hash_insert_wildcard(connf_t *, conn_t *);
void ipcl_hash_remove(conn_t *);
void ipcl_hash_remove_locked(conn_t *connp, connf_t *connfp);

extern int      ipcl_bind_insert(conn_t *);
extern int      ipcl_bind_insert_v4(conn_t *);
extern int      ipcl_bind_insert_v6(conn_t *);
extern int      ipcl_conn_insert(conn_t *);
extern int      ipcl_conn_insert_v4(conn_t *);
extern int      ipcl_conn_insert_v6(conn_t *);
extern conn_t   *ipcl_get_next_conn(connf_t *, conn_t *, uint32_t);

conn_t *ipcl_classify_v4(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,
            ip_stack_t *);
conn_t *ipcl_classify_v6(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,
            ip_stack_t *);
conn_t *ipcl_classify(mblk_t *, ip_recv_attr_t *, ip_stack_t *);
conn_t *ipcl_classify_raw(mblk_t *, uint8_t, uint32_t, ipha_t *,
    ip6_t *, ip_recv_attr_t *, ip_stack_t *);
conn_t *ipcl_iptun_classify_v4(ipaddr_t *, ipaddr_t *, ip_stack_t *);
conn_t *ipcl_iptun_classify_v6(in6_addr_t *, in6_addr_t *, ip_stack_t *);
void    ipcl_globalhash_insert(conn_t *);
void    ipcl_globalhash_remove(conn_t *);
```



```
void    ipcl_walk(pfv_t, void *, ip_stack_t *);
conn_t  *ipcl_tcp_lookup_reversed_ipv4(ipha_t *, tcpha_t *, int, ip_stack_t *);
conn_t  *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpha_t *, int, uint_t,
            ip_stack_t *);
conn_t  *ipcl_lookup_listener_v4(uint16_t, ipaddr_t, zoneid_t, ip_stack_t *);
conn_t  *ipcl_lookup_listener_v6(uint16_t, in6_addr_t *, uint_t, zoneid_t,
            ip_stack_t *);
int     conn_trace_ref(conn_t *);
int     conn_untrace_ref(conn_t *);
void    ipcl_conn_cleanup(conn_t *);
extern uint_t   conn_recvancillary_size(conn_t *, crb_t, ip_recv_attr_t *,
    mblk_t *, ip_pkt_t *);
extern void     conn_recvancillary_add(conn_t *, crb_t, ip_recv_attr_t *,
    ip_pkt_t *, uchar_t *, uint_t);
conn_t *ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *, ipha_t *, tcpha_t *,
            ip_stack_t *);
conn_t *ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *, ip6_t *, tcpha_t *,
            ip_stack_t *);

extern int ip_create_helper_stream(conn_t *, ldi_ident_t);
extern void ip_free_helper_stream(conn_t *);
extern int      ip_helper_stream_setup(queue_t *, dev_t *, int, int,
    cred_t *, boolean_t);

#ifdef  __cplusplus
}
#endif

#endif  /* _INET_IPCLASSIFIER_H */
```

```
**********************************************************
   12863 Mon Jul  9 14:38:19 2012
new/usr/src/uts/common/inet/kstatcom.h
dccp: stats
**********************************************************
______unchanged_portion_omitted_

typedef struct dccp_named_kstat {
        kstat_named_t activeOpens;
        kstat_named_t passiveOpens;
        kstat_named_t inSegs;
        kstat_named_t outSegs;
} dccp_named_kstat_t;
#endif /* ! codereview */

#define NUM_OF_FIELDS(S)        (sizeof (S) / sizeof (kstat_named_t))

#ifdef  __cplusplus
}
#endif

#endif  /* _INET_KSTATCOM_H */
```

```
**********************************************************
   62568 Mon Jul  9 14:38:19 2012
new/usr/src/uts/common/inet/mib2.h
dccp: MIB-II
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 *
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/* Copyright (c) 1990 Mentat Inc. */

#ifndef _INET_MIB2_H
#define _INET_MIB2_H

#include <netinet/in.h> /* For in6_addr_t */
#include <sys/tsol/label.h> /* For brange_t */
#include <sys/tsol/label_macro.h> /* For brange_t */

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * The IPv6 parts of this are derived from:
 *      RFC 2465
 *      RFC 2466
 *      RFC 2452
 *      RFC 2454
 */

/*
 * SNMP set/get via M_PROTO T_OPTMGMT_REQ.  Structure is that used
 * for [gs]etsockopt() calls.  get uses T_CURRENT, set uses T_NEOGTIATE
 * MGMT_flags value.  The following definition of opthdr is taken from
 * socket.h:
 *
 * An option specification consists of an opthdr, followed by the value of
 * the option.  An options buffer contains one or more options.  The len
 * field of opthdr specifies the length of the option value in bytes.  This
 * length must be a multiple of sizeof(long) (use OPTLEN macro).
 *
 * struct opthdr {
 *      long    level;  protocol level affected
 *      long    name;   option to modify
 *      long    len;    length of option value
 * };
 *
 * #define OPTLEN(x) ((((x) + sizeof(long) - 1) / sizeof(long)) * sizeof(long))
```



```
 * #define OPTVAL(opt) ((char *)(opt + 1))
 *
 * For get requests (T_CURRENT), any MIB2_xxx value can be used (only
 * "get all" is supported, so all modules get a copy of the request to
 * return everything it knows.   In general, we use MIB2_IP.  There is
 * one exception: in general, IP will not report information related to
 * ire_testhidden and IRE_IF_CLONE routes (e.g., in the MIB2_IP_ROUTE
 * table). However, using the special value EXPER_IP_AND_ALL_IRES will cause
 * all information to be reported.  This special value should only be
 * used by IPMP-aware low-level utilities (e.g. in.mpathd).
 *
 * IMPORTANT:  some fields are grouped in a different structure than
 * suggested by MIB-II, e.g., checksum error counts.  The original MIB-2
 * field name has been retained.  Field names beginning with "mi" are not
 * defined in the MIB but contain important & useful information maintained
 * by the corresponding module.
 */
#ifndef IPPROTO_MAX
#define IPPROTO_MAX     256
#endif

#define MIB2_SYSTEM             (IPPROTO_MAX+1)
#define MIB2_INTERFACES         (IPPROTO_MAX+2)
#define MIB2_AT                 (IPPROTO_MAX+3)
#define MIB2_IP                 (IPPROTO_MAX+4)
#define MIB2_ICMP               (IPPROTO_MAX+5)
#define MIB2_TCP                (IPPROTO_MAX+6)
#define MIB2_UDP                (IPPROTO_MAX+7)
#define MIB2_EGP                (IPPROTO_MAX+8)
#define MIB2_CMOT               (IPPROTO_MAX+9)
#define MIB2_TRANSMISSION       (IPPROTO_MAX+10)
#define MIB2_SNMP               (IPPROTO_MAX+11)
#define MIB2_IP6                (IPPROTO_MAX+12)
#define MIB2_ICMP6              (IPPROTO_MAX+13)
#define MIB2_TCP6               (IPPROTO_MAX+14)
#define MIB2_UDP6               (IPPROTO_MAX+15)
#define MIB2_SCTP               (IPPROTO_MAX+16)
#define MIB2_DCCP               (IPPROTO_MAX+17)
#define MIB2_DCCP6              (IPPROTO_MAX+18)
#endif /* ! codereview */

/*
 * Define range of levels for use with MIB2_*
 */
#define MIB2_RANGE_START        (IPPROTO_MAX+1)
#define MIB2_RANGE_END          (IPPROTO_MAX+18)
#define MIB2_RANGE_END          (IPPROTO_MAX+16)
```

```
#define EXPER                   1024    /* experimental - not part of mib */
#define EXPER_IGMP              (EXPER+1)
#define EXPER_DVMRP             (EXPER+2)
#define EXPER_RAWIP             (EXPER+3)
#define EXPER_IP_AND_ALL_IRES   (EXPER+4)

/*
 * Define range of levels for experimental use
 */
#define EXPER_RANGE_START       (EXPER+1)
#define EXPER_RANGE_END         (EXPER+4)

#define BUMP_MIB(s, x)          {                                       \
        extern void __dtrace_probe___mib_##x(int, void *);              \
        void *stataddr = &((s)->x);                                     \
        __dtrace_probe___mib_##x(1, stataddr);                          \
        (s)->x++;                                                       \
```

```
 127 }
_____unchanged_portion_omitted_

1788 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1789 #pragma pack()
1790 #endif

1792 /*
1793  * the DCCP group
1794  */
1795 #define MIB2_DCCP_CONN          18
1796 #define MIB2_DCCP6_CONN         19

1798 #define MIB2_DCCP_closed        1
1799 #define MIB2_DCCP_listen        2

1801 /* Pack data to make struct size the same for 32- and 64-bits */
1802 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1803 #pragma pack(4)
1804 #endif

1806 typedef struct mib2_dccp {

1808                 /* # of direct transitions CLOSED -> ACK-SENT   { dccp 5 } */
1809         Counter dccpActiveOpens;
1810                 /* # of direct transitions LISTEN -> ACK-RCVD   { dccp 6 } */
1811         Counter dccpPassiveOpens;
1812                 /* # of direct SIN-SENT/RCVD -> CLOSED/LISTEN   { dccp 7 } */
1813         Counter dccpAttemptFails;
1814                 /* # of direct ESTABLISHED/CLOSE-WAIT -> CLOSED { dccp 8 } */
1815         Counter dccpEstabResets;
1816                 /* # of connections ESTABLISHED or CLOSE-WAIT   { dccp 9 } */
1817         Gauge   dccpCurrEstab;
1818                 /* total # of segments recv'd                   { dccp 10 } */
1819         Counter dccpInSegs;
1820                 /* total # of segments sent                     { dccp 11 } */
1821         Counter dccpOutSegs;
1822                 /* total # of segments retransmitted            { dccp 12 } */
1823         Counter dccpRetransSegs;

1825         int     dccpEntrySize;
1826         int     dccp6EntrySize;

1828         int     dccpConnTableSize;
1829         int     dccp6ConnTableSize;

1831         Counter64       dccpHCInDatagrams;
1832 } mib2_dccp_t;
1833 #define MIB_FIRST_NEW_ELM_mib2_dccp_t   dccpHCInDatagrams

1835 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1836 #pragma pack()
1837 #endif

1839 /* Pack data to make struct size the same for 32- and 64-bits */
1840 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1841 #pragma pack(4)
1842 #endif

1844 typedef struct mib2_dccpConnEntry {
1845         int             dccpConnState;
1846         IpAddress       dccpConnLocalAddress;
1847         int             dccpConnLocalPort;
1848         IpAddress       dccpConnRemAddress;
1849         int             dccpConnRemPort;
```



```
1851         uint32_t        dccpConnCreationProcess;
1852         uint64_t        dccpConnCreationTime;
1853 } mib2_dccpConnEntry_t;
1854 #define MIB_FIRST_NEW_ELM_mib2_dccpConnEntry_t  dccpConnCreationProcess

1856 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1857 #pragma pack()
1858 #endif

1860 /* Pack data to make struct size the same for 32- and 64-bits */
1861 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1862 #pragma pack(4)
1863 #endif

1865 typedef struct mib2_dccp6ConnEntry {
1866         int             dccpConnState;
1867         Ip6Address      dccpConnLocalAddress;
1868         int             dccpConnLocalPort;
1869         Ip6Address      dccpConnRemAddress;
1870         int             dccpConnRemPort;

1872         uint32_t        dccpConnCreationProcess;
1873         uint64_t        dccpConnCreationTime;
1874 } mib2_dccp6ConnEntry_t;
1875 #define MIB_FIRST_NEW_ELM_mib2_dccp6ConnEntry_t dccpConnCreationProcess

1877 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1878 #pragma pack()
1879 #endif
1880 #endif /* ! codereview */

1882 #ifdef  __cplusplus
1883 }
1884 #endif

1886 #endif  /* _INET_MIB2_H */
```

```
**********************************************************
   46352 Mon Jul  9 14:38:19 2012
new/usr/src/uts/common/inet/sctp/sctp_impl.h
sctp: align to cache line
**********************************************************
______unchanged_portion_omitted_

 428 /*
 429  * Bind hash array size and hash function.  The size must be a power
 430  * of 2 and lport must be in host byte order.
 431  */
 432 #define SCTP_BIND_FANOUT_SIZE   2048
 433 #define SCTP_BIND_HASH(lport)   (((lport) * 31) & (SCTP_BIND_FANOUT_SIZE - 1))

 435 /* options that SCTP negotiates during association establishment */
 436 #define SCTP_PRSCTP_OPTION      0x01

 438 /*
 439  * Listener hash array size and hash function.  The size must be a power
 440  * of 2 and lport must be in host byte order.
 441  */
 442 #define SCTP_LISTEN_FANOUT_SIZE 512
 443 #define SCTP_LISTEN_HASH(lport) (((lport) * 31) & (SCTP_LISTEN_FANOUT_SIZE - 1))

 445 typedef struct sctp_tf_s {
 446         struct sctp_s   *tf_sctp;
 447         kmutex_t        tf_lock;
 448 #define SF_CACHEL_PAD   64
 449         uchar_t         tf_pad[SF_CACHEL_PAD - (sizeof (struct sctp_s *) +
 450                             sizeof (kmutex_t))];
 451 #endif /* ! codereview */
 452 } sctp_tf_t;

 454 /* Round up the value to the nearest mss. */
 455 #define MSS_ROUNDUP(value, mss)         ((((value) - 1) / (mss) + 1) * (mss))

 457 extern sin_t    sctp_sin_null;  /* Zero address for quick clears */
 458 extern sin6_t   sctp_sin6_null; /* Zero address for quick clears */

 460 #define SCTP_IS_DETACHED(sctp)          ((sctp)->sctp_detached)

 462 /* Data structure used to track received TSNs */
 463 typedef struct sctp_set_s {
 464         struct sctp_set_s *next;
 465         struct sctp_set_s *prev;
 466         uint32_t begin;
 467         uint32_t end;
 468 } sctp_set_t;

 470 /* Data structure used to track TSNs for PR-SCTP */
 471 typedef struct sctp_ftsn_set_s {
 472         struct sctp_ftsn_set_s *next;
 473         ftsn_entry_t    ftsn_entries;
 474 } sctp_ftsn_set_t;

 476 /* Data structure used to track incoming SCTP streams */
 477 typedef struct sctp_instr_s {
 478         mblk_t          *istr_msgs;
 479         int             istr_nmsgs;
 480         uint16_t        nextseq;
 481         struct sctp_s   *sctp;
 482         mblk_t          *istr_reass;
 483 } sctp_instr_t;

 485 /* Reassembly data structure (per-stream) */
 486 typedef struct sctp_reass_s {
```



```
 487         uint16_t        sr_ssn;
 488         uint16_t        sr_needed;
 489         uint16_t        sr_got;
 490         uint16_t        sr_msglen;      /* len of consecutive fragments */
 491                                         /* from the begining (B-bit) */
 492         mblk_t          *sr_tail;
 493         boolean_t       sr_hasBchunk;   /* If the fragment list begins with */
 494                                         /* a B-bit set chunk */
 495         uint32_t        sr_nexttsn;     /* TSN of the next fragment we */
 496                                         /* are expecting */
 497         boolean_t       sr_partial_delivered;
 498 } sctp_reass_t;

 500 /* debugging */
 501 #undef  dprint
 502 #ifdef DEBUG
 503 extern int sctpdebug;
 504 #define dprint(level, args)     { if (sctpdebug > (level)) printf args; }
 505 #else
 506 #define dprint(level, args) {}
 507 #endif


 510 /* Peer address tracking */

 512 /*
 513  * States for peer addresses
 514  *
 515  * SCTP_FADDRS_UNCONFIRMED: we have not communicated with this peer address
 516  *     before, mark it as unconfirmed so that we will not send data to it.
 517  *     All addresses initially are in unconfirmed state and required
 518  *     validation.  SCTP sends a heartbeat to each of them and when it gets
 519  *     back a heartbeat ACK, the address will be marked as alive.  This
 520  *     validation fixes a security issue with multihoming.  If an attacker
 521  *     establishes an association with us and tells us that it has addresses
 522  *     belonging to another host A, this will prevent A from communicating
 523  *     with us.  This is fixed by peer address validation.  In the above case,
 524  *     A will respond with an abort.
 525  *
 526  * SCTP_FADDRS_ALIVE: this peer address is alive and we can communicate with
 527  *     it with no problem.
 528  *
 529  * SCTP_FADDRS_DOWN: we have exceeded the retransmission limit to this
 530  *     peer address.  Once an address is marked down, we will only send
 531  *     a heartbeat to it every hb_interval in case it becomes alive now.
 532  *
 533  * SCTP_FADDRS_UNREACH: there is no suitable source address to send to
 534  *     this peer address.  For example, the peer address is v6 but we only
 535  *     have v4 addresses.  It is marked unreachable until there is an
 536  *     address configuration change.  At that time, mark these addresses
 537  *     as unconfirmed and try again to see if those unreachable addresses
 538  *     are OK as we may have more source addresses.
 539  */
 540 typedef enum {
 541         SCTP_FADDRS_UNREACH,
 542         SCTP_FADDRS_DOWN,
 543         SCTP_FADDRS_ALIVE,
 544         SCTP_FADDRS_UNCONFIRMED
 545 } faddr_state_t;

 547 typedef struct sctp_faddr_s {
 548         struct sctp_faddr_s *sf_next;
 549         faddr_state_t   sf_state;

 551         in6_addr_t      sf_faddr;
 552         in6_addr_t      sf_saddr;
```

```
554         int64_t         sf_hb_expiry;   /* time to retransmit heartbeat */
555         uint32_t        sf_hb_interval; /* the heartbeat interval */

557         int             sf_rto;         /* RTO in tick */
558         int             sf_srtt;        /* Smoothed RTT in tick */
559         int             sf_rttvar;      /* RTT variance in tick */
560         uint32_t        sf_rtt_updates;
561         int             sf_strikes;
562         int             sf_max_retr;
563         uint32_t        sf_pmss;
564         uint32_t        sf_cwnd;
565         uint32_t        sf_ssthresh;
566         uint32_t        sf_suna;        /* sent - unack'ed */
567         uint32_t        sf_pba;         /* partial bytes acked */
568         uint32_t        sf_acked;
569         int64_t         sf_lastactive;
570         mblk_t          *sf_timer_mp;   /* retransmission timer control */
571         uint32_t
572                         sf_hb_pending : 1,
573                         sf_timer_running : 1,
574                         sf_df : 1,
575                         sf_pmtu_discovered : 1,

577                         sf_rc_timer_running : 1,
578                         sf_isv4 : 1,
579                         sf_hb_enabled : 1;

581         mblk_t          *sf_rc_timer_mp; /* reliable control chunk timer */
582         ip_xmit_attr_t  *sf_ixa;        /* Transmit attributes */
583         uint32_t        sf_T3expire;    /* # of times T3 timer expired */

585         uint64_t        sf_hb_secret;   /* per addr "secret" in heartbeat */
586         uint32_t        sf_rxt_unacked; /* # unack'ed retransmitted bytes */
587 } sctp_faddr_t;

589 /* Flags to indicate supported address type in the PARM_SUP_ADDRS. */
590 #define PARM_SUPP_V6    0x1
591 #define PARM_SUPP_V4    0x2

593 /*
594  * Set heartbeat interval plus jitter.  The jitter is supposed to be random,
595  * up to +/- 50% of the RTO.  We use gethrtime() here for  performance reason
596  * as the jitter does not really need to be "very" random.
597  */
598 #define SET_HB_INTVL(fp)                                                \
599         ((fp)->sf_hb_interval + (fp)->sf_rto + ((fp)->sf_rto >> 1) -    \
600         (uint_t)gethrtime() % (fp)->sf_rto)

602 #define SCTP_IPIF_HASH  16

604 typedef struct  sctp_ipif_hash_s {
605         list_t          sctp_ipif_list;
606         int             ipif_count;
607         krwlock_t       ipif_hash_lock;
608 } sctp_ipif_hash_t;


611 /*
612  * Initialize cwnd according to RFC 3390.  def_max_init_cwnd is
613  * either sctp_slow_start_initial or sctp_slow_start_after idle
614  * depending on the caller.
615  */
616 #define SET_CWND(fp, mss, def_max_init_cwnd)                            \
617 {                                                                       \
618         (fp)->sf_cwnd = MIN(def_max_init_cwnd * (mss),                  \
```



```
619             MIN(4 * (mss), MAX(2 * (mss), 4380 / (mss) * (mss))));      \
620 }


623 struct sctp_s;

625 /*
626  * Control structure for each open SCTP stream,
627  * defined only within the kernel or for a kmem user.
628  * NOTE: sctp_reinit_values MUST have a line for each field in this structure!
629  */
630 #if (defined(_KERNEL) || defined(_KMEMUSER))

632 typedef struct sctp_s {

634         /*
635          * The following is shared with (and duplicated) in IP, so if you
636          * make changes, make sure you also change things in ip_sctp.c.
637          */
638         struct sctp_s   *sctp_conn_hash_next;
639         struct sctp_s   *sctp_conn_hash_prev;

641         struct sctp_s   *sctp_listen_hash_next;
642         struct sctp_s   *sctp_listen_hash_prev;

644         sctp_tf_t       *sctp_listen_tfp;       /* Ptr to tf */
645         sctp_tf_t       *sctp_conn_tfp;         /* Ptr to tf */

647         /* Global list of sctp */
648         list_node_t     sctp_list;

650         sctp_faddr_t            *sctp_faddrs;
651         int                     sctp_nfaddrs;
652         sctp_ipif_hash_t        sctp_saddrs[SCTP_IPIF_HASH];
653         int                     sctp_nsaddrs;

655         kmutex_t        sctp_lock;
656         kcondvar_t      sctp_cv;
657         boolean_t       sctp_running;

659 #define sctp_ulpd       sctp_connp->conn_upper_handle
660 #define sctp_upcalls    sctp_connp->conn_upcalls

662 #define sctp_ulp_newconn        sctp_upcalls->su_newconn
663 #define sctp_ulp_connected      sctp_upcalls->su_connected
664 #define sctp_ulp_disconnected   sctp_upcalls->su_disconnected
665 #define sctp_ulp_opctl          sctp_upcalls->su_opctl
666 #define sctp_ulp_recv           sctp_upcalls->su_recv
667 #define sctp_ulp_txq_full       sctp_upcalls->su_txq_full
668 #define sctp_ulp_prop           sctp_upcalls->su_set_proto_props

670         int32_t         sctp_state;

672         conn_t          *sctp_connp;            /* conn_t stuff */
673         sctp_stack_t    *sctp_sctps;

675         /* Peer address tracking */
676         sctp_faddr_t    *sctp_lastfaddr;        /* last faddr in list */
677         sctp_faddr_t    *sctp_primary;          /* primary faddr */
678         sctp_faddr_t    *sctp_current;          /* current faddr */
679         sctp_faddr_t    *sctp_lastdata;         /* last data seen from this */

681         /* Outbound data tracking */
682         mblk_t          *sctp_xmit_head;
683         mblk_t          *sctp_xmit_tail;
684         mblk_t          *sctp_xmit_unsent;
```

```
685         mblk_t          *sctp_xmit_unsent_tail;
686         mblk_t          *sctp_xmit_unacked;

688         int32_t         sctp_unacked;           /* # of unacked bytes */
689         int32_t         sctp_unsent;            /* # of unsent bytes in hand */

691         uint32_t        sctp_ltsn;              /* Local instance TSN */
692         uint32_t        sctp_lastack_rxd;       /* Last rx'd cumtsn */
693         uint32_t        sctp_recovery_tsn;      /* Exit from fast recovery */
694         uint32_t        sctp_adv_pap;           /* Adv. Peer Ack Point */

696         uint16_t        sctp_num_ostr;
697         uint16_t        *sctp_ostrcntrs;

699         mblk_t          *sctp_pad_mp;           /* pad unaligned data chunks */

701         /* sendmsg() default parameters */
702         uint16_t        sctp_def_stream;        /* default stream id */
703         uint16_t        sctp_def_flags;         /* default xmit flags */
704         uint32_t        sctp_def_ppid;          /* default payload id */
705         uint32_t        sctp_def_context;       /* default context */
706         uint32_t        sctp_def_timetolive;    /* default msg TTL */

708         /* Inbound data tracking */
709         sctp_set_t      *sctp_sack_info;        /* Sack tracking */
710         mblk_t          *sctp_ack_mp;           /* Delayed ACK timer block */
711         sctp_instr_t    *sctp_instr;            /* Instream trackers */
712         mblk_t          *sctp_uo_frags;         /* Un-ordered msg. fragments */
713         uint32_t        sctp_ftsn;              /* Peer's TSN */
714         uint32_t        sctp_lastacked;         /* last cumtsn SACKd */
715         uint16_t        sctp_num_istr;          /* No. of instreams */
716         int32_t         sctp_istr_nmsgs;        /* No. of chunks in instreams */
717         int32_t         sctp_sack_gaps;         /* No. of received gaps */
718         int32_t         sctp_sack_toggle;       /* SACK every other pkt */

720         /* RTT calculation */
721         uint32_t        sctp_rtt_tsn;
722         int64_t         sctp_out_time;

724         /* Stats can be reset by snmp users kstat, netstat and snmp agents */
725         uint64_t        sctp_opkts;             /* sent pkts */
726         uint64_t        sctp_obchunks;          /* sent control chunks */
727         uint64_t        sctp_odchunks;          /* sent ordered data chunks */
728         uint64_t        sctp_oudchunks;         /* sent unord data chunks */
729         uint64_t        sctp_rxtchunks;         /* retransmitted chunks */
730         uint64_t        sctp_ipkts;             /* recv pkts */
731         uint64_t        sctp_ibchunks;          /* recv control chunks */
732         uint64_t        sctp_idchunks;          /* recv ordered data chunks */
733         uint64_t        sctp_iudchunks;         /* recv unord data chunks */
734         uint64_t        sctp_fragdmsgs;
735         uint64_t        sctp_reassmsgs;
736         uint32_t        sctp_T1expire;          /* # of times T1timer expired */
737         uint32_t        sctp_T2expire;          /* # of times T2timer expired */
738         uint32_t        sctp_T3expire;          /* # of times T3timer expired */
739         uint32_t        sctp_assoc_start_time;  /* time when assoc was est. */

741         uint32_t        sctp_frwnd;             /* Peer RWND */
742         uint32_t        sctp_cwnd_max;

744         /* Inbound flow control */
745         int32_t         sctp_rwnd;              /* Current receive window */
746         int32_t         sctp_arwnd;             /* Last advertised window */
747         int32_t         sctp_rxqueued;          /* No. of bytes in RX q's */
748         int32_t         sctp_ulp_rxqueued;      /* Data in ULP */

750         /* Pre-initialized composite headers */
```



```
751         uchar_t         *sctp_iphc;     /* v4 sctp/ip hdr template buffer */
752         uchar_t         *sctp_iphc6;    /* v6 sctp/ip hdr template buffer */

754         int32_t         sctp_iphc_len;  /* actual allocated v4 buffer size */
755         int32_t         sctp_iphc6_len; /* actual allocated v6 buffer size */

757         int32_t         sctp_hdr_len;   /* len of combined SCTP/IP v4 hdr */
758         int32_t         sctp_hdr6_len;  /* len of combined SCTP/IP v6 hdr */

760         ipha_t          *sctp_ipha;     /* IPv4 header in the buffer */
761         ip6_t           *sctp_ip6h;     /* IPv6 header in the buffer */

763         int32_t         sctp_ip_hdr_len; /* Byte len of our current v4 hdr */
764         int32_t         sctp_ip_hdr6_len; /* Byte len of our current v6 hdr */

766         sctp_hdr_t      *sctp_sctph;    /* sctp header in combined v4 hdr */
767         sctp_hdr_t      *sctp_sctph6;   /* sctp header in combined v6 hdr */

769         uint32_t        sctp_lvtag;     /* local SCTP instance verf tag */
770         uint32_t        sctp_fvtag;     /* Peer's SCTP verf tag */

772         /* Path MTU Discovery */
773         int64_t         sctp_last_mtu_probe;
774         clock_t         sctp_mtu_probe_intvl;
775         uint32_t        sctp_mss;       /* Max send size (not TCP MSS!) */

777         /* structs sctp_bits, sctp_events are for clearing all bits at once */
778         struct {
779                 uint32_t

781                 sctp_understands_asconf : 1, /* Peer handles ASCONF chunks */
782                 sctp_cchunk_pend : 1,   /* Control chunk in flight. */
783                 sctp_lingering : 1,     /* Lingering in close */
784                 sctp_loopback: 1,       /* src and dst are the same machine */

786                 sctp_force_sack : 1,
787                 sctp_ack_timer_running: 1,      /* Delayed ACK timer running */
788                 sctp_hwcksum : 1,       /* The NIC is capable of hwcksum */
789                 sctp_understands_addip : 1,

791                 sctp_bound_to_all : 1,
792                 sctp_cansleep : 1,      /* itf routines can sleep */
793                 sctp_detached : 1,      /* If we're detached from a stream */
794                 sctp_send_adaptation : 1,       /* send adaptation layer ind */

796                 sctp_recv_adaptation : 1,       /* recv adaptation layer ind */
797                 sctp_ndelay : 1,        /* turn off Nagle */
798                 sctp_condemned : 1,     /* this sctp is about to disappear */
799                 sctp_chk_fast_rexmit : 1, /* check for fast rexmit message */

801                 sctp_prsctp_aware : 1,  /* is peer PR-SCTP aware? */
802                 sctp_linklocal : 1,     /* is linklocal assoc. */
803                 sctp_rexmitting : 1,    /* SCTP is retransmitting */
804                 sctp_zero_win_probe : 1,        /* doing zero win probe */

806                 sctp_txq_full : 1,      /* the tx queue is full */
807                 sctp_ulp_discon_done : 1,       /* ulp_disconnecting done */
808                 sctp_flowctrld : 1,     /* upper layer flow controlled */
809                 sctp_dummy : 5;
810         } sctp_bits;
811         struct {
812                 uint32_t

814                 sctp_recvsndrcvinfo : 1,
815                 sctp_recvassocevnt : 1,
816                 sctp_recvpathevnt : 1,
```

```
817                 sctp_recvsendfailevnt : 1,

819                 sctp_recvpeererr : 1,
820                 sctp_recvshutdownevnt : 1,
821                 sctp_recvpdevnt : 1,
822                 sctp_recvalevnt : 1;
823         } sctp_events;
824 #define sctp_priv_stream sctp_bits.sctp_priv_stream
825 #define sctp_understands_asconf sctp_bits.sctp_understands_asconf
826 #define sctp_cchunk_pend sctp_bits.sctp_cchunk_pend
827 #define sctp_lingering sctp_bits.sctp_lingering
828 #define sctp_loopback sctp_bits.sctp_loopback
829 #define sctp_force_sack sctp_bits.sctp_force_sack
830 #define sctp_ack_timer_running sctp_bits.sctp_ack_timer_running
831 #define sctp_hwcksum sctp_bits.sctp_hwcksum
832 #define sctp_understands_addip sctp_bits.sctp_understands_addip
833 #define sctp_bound_to_all sctp_bits.sctp_bound_to_all
834 #define sctp_cansleep sctp_bits.sctp_cansleep
835 #define sctp_detached sctp_bits.sctp_detached
836 #define sctp_send_adaptation sctp_bits.sctp_send_adaptation
837 #define sctp_recv_adaptation sctp_bits.sctp_recv_adaptation
838 #define sctp_ndelay sctp_bits.sctp_ndelay
839 #define sctp_condemned sctp_bits.sctp_condemned
840 #define sctp_chk_fast_rexmit sctp_bits.sctp_chk_fast_rexmit
841 #define sctp_prsctp_aware sctp_bits.sctp_prsctp_aware
842 #define sctp_linklocal sctp_bits.sctp_linklocal
843 #define sctp_rexmitting sctp_bits.sctp_rexmitting
844 #define sctp_zero_win_probe sctp_bits.sctp_zero_win_probe
845 #define sctp_txq_full sctp_bits.sctp_txq_full
846 #define sctp_ulp_discon_done sctp_bits.sctp_ulp_discon_done
847 #define sctp_flowctrld sctp_bits.sctp_flowctrld

849 #define sctp_recvsndrcvinfo sctp_events.sctp_recvsndrcvinfo
850 #define sctp_recvassocevnt sctp_events.sctp_recvassocevnt
851 #define sctp_recvpathevnt sctp_events.sctp_recvpathevnt
852 #define sctp_recvsendfailevnt sctp_events.sctp_recvsendfailevnt
853 #define sctp_recvpeererr sctp_events.sctp_recvpeererr
854 #define sctp_recvshutdownevnt sctp_events.sctp_recvshutdownevnt
855 #define sctp_recvpdevnt sctp_events.sctp_recvpdevnt
856 #define sctp_recvalevnt sctp_events.sctp_recvalevnt

858         /* Retransmit info */
859         mblk_t          *sctp_cookie_mp; /* cookie chunk, if rxt needed */
860         int32_t         sctp_strikes;   /* Total number of assoc strikes */
861         int32_t         sctp_max_init_rxt;
862         int32_t         sctp_pa_max_rxt; /* Max per-assoc retransmit cnt */
863         int32_t         sctp_pp_max_rxt; /* Max per-path retransmit cnt */
864         uint32_t        sctp_rto_max;
865         uint32_t        sctp_rto_max_init;
866         uint32_t        sctp_rto_min;
867         uint32_t        sctp_rto_initial;

869         int64_t         sctp_last_secret_update;
870         uint8_t         sctp_secret[SCTP_SECRET_LEN]; /* for cookie auth */
871         uint8_t         sctp_old_secret[SCTP_SECRET_LEN];
872         uint32_t        sctp_cookie_lifetime;   /* cookie lifetime in tick */

874         /* Bind hash tables */
875         kmutex_t        *sctp_bind_lockp;       /* Ptr to tf_lock */
876         struct sctp_s   *sctp_bind_hash;
877         struct sctp_s **sctp_ptpbhn;

879         /* Shutdown / cleanup */
880         sctp_faddr_t    *sctp_shutdown_faddr;   /* rotate faddr during shutd */
881         int32_t         sctp_client_errno;      /* How the client screwed up */
882         kmutex_t        sctp_reflock;   /* Protects sctp_refcnt & timer mp */
```



```
883         ushort_t        sctp_refcnt;    /* No. of pending upstream msg */
884         mblk_t          *sctp_timer_mp; /* List of fired timers. */

886         mblk_t          *sctp_heartbeat_mp; /* Timer block for heartbeats */
887         uint32_t        sctp_hb_interval; /* Default hb_interval */

889         int32_t         sctp_autoclose; /* Auto disconnect in ticks */
890         int64_t         sctp_active;    /* Last time data/sack on this conn */
891         uint32_t        sctp_tx_adaptation_code; /* TX adaptation code */
892         uint32_t        sctp_rx_adaptation_code; /* RX adaptation code */

894         /* Reliable control chunks */
895         mblk_t          *sctp_cxmit_list; /* Xmit list for control chunks */
896         uint32_t        sctp_lcsn;      /* Our serial number */
897         uint32_t        sctp_fcsn;      /* Peer serial number */

899         /* Per association receive queue */
900         kmutex_t        sctp_recvq_lock;
901         mblk_t          *sctp_recvq;
902         mblk_t          *sctp_recvq_tail;
903         taskq_t         *sctp_recvq_tq;

905         /* IPv6 ancillary data */
906         uint_t          sctp_recvifindex;       /* last rcvd IPV6_RCVPKTINFO */
907         uint_t          sctp_recvhops;          /*  " IPV6_RECVHOPLIMIT */
908         uint_t          sctp_recvtclass;        /*  " IPV6_RECVTCLASS */
909         ip6_hbh_t       *sctp_hopopts;          /*  " IPV6_RECVHOPOPTS */
910         ip6_dest_t      *sctp_dstopts;          /*  " IPV6_RECVDSTOPTS */
911         ip6_dest_t      *sctp_rthdrdstopts;     /*  " IPV6_RECVRTHDRDSTOPTS */
912         ip6_rthdr_t     *sctp_rthdr;            /*  " IPV6_RECVRTHDR */
913         uint_t          sctp_hopoptslen;
914         uint_t          sctp_dstoptslen;
915         uint_t          sctp_rthdrdstoptslen;
916         uint_t          sctp_rthdrlen;

918         /* Stats */
919         uint64_t        sctp_msgcount;
920         uint64_t        sctp_prsctpdrop;

922         uint_t          sctp_v4label_len;       /* length of cached v4 label */
923         uint_t          sctp_v6label_len;       /* length of cached v6 label */
924         uint32_t        sctp_rxt_nxttsn;        /* Next TSN to be rexmitted */
925         uint32_t        sctp_rxt_maxtsn;        /* Max TSN sent at time out */

927         int             sctp_pd_point;          /* Partial delivery point */
928         mblk_t          *sctp_err_chunks;       /* Error chunks */
929         uint32_t        sctp_err_len;           /* Total error chunks length */

931         /* additional source data for per endpoint association statistics */
932         uint64_t        sctp_outseqtsns;        /* TSN rx > expected TSN */
933         uint64_t        sctp_osacks;            /* total sacks sent */
934         uint64_t        sctp_isacks;            /* total sacks received */
935         uint64_t        sctp_idupchunks;        /* rx dups, ord or unord */
936         uint64_t        sctp_gapcnt;            /* total gap acks rx */
937         /*
938          * Add the current data from the counters which are reset by snmp
939          * to these cumulative counters to use in per endpoint statistics.
940          */
941         uint64_t        sctp_cum_obchunks;      /* sent control chunks */
942         uint64_t        sctp_cum_odchunks;      /* sent ordered data chunks */
943         uint64_t        sctp_cum_oudchunks;     /* sent unord data chunks */
944         uint64_t        sctp_cum_rxtchunks;     /* retransmitted chunks */
945         uint64_t        sctp_cum_ibchunks;      /* recv control chunks */
946         uint64_t        sctp_cum_idchunks;      /* recv ordered data chunks */
947         uint64_t        sctp_cum_iudchunks;     /* recv unord data chunks */
```

```
949         /*
950          * When non-zero, this is the maximum observed RTO since assoc stats
951          * were last requested. When zero, no RTO update has occurred since
952          * the previous user request for stats on this endpoint.
953          */
954         int     sctp_maxrto;
955         /*
956          * The stored value of sctp_maxrto passed to user during the previous
957          * user request for stats on this endpoint.
958          */
959         int     sctp_prev_maxrto;

961         /* For association counting. */
962         sctp_listen_cnt_t       *sctp_listen_cnt;
963 } sctp_t;

965 #define SCTP_TXQ_LEN(sctp)      ((sctp)->sctp_unsent + (sctp)->sctp_unacked)
966 #define SCTP_TXQ_UPDATE(sctp)                                   \
967         if ((sctp)->sctp_txq_full && SCTP_TXQ_LEN(sctp) <=      \
968             (sctp)->sctp_connp->conn_sndlowat) {                \
969                 (sctp)->sctp_txq_full = 0;                      \
970                 (sctp)->sctp_ulp_txq_full((sctp)->sctp_ulpd,    \
971                     B_FALSE);                                   \
972         }

974 #endif  /* (defined(_KERNEL) || defined(_KMEMUSER)) */

976 extern void     sctp_ack_timer(sctp_t *);
977 extern size_t   sctp_adaptation_code_param(sctp_t *, uchar_t *);
978 extern void     sctp_adaptation_event(sctp_t *);
979 extern void     sctp_add_err(sctp_t *, uint16_t, void *, size_t,
980                     sctp_faddr_t *);
981 extern int      sctp_add_faddr(sctp_t *, in6_addr_t *, int, boolean_t);
982 extern boolean_t sctp_add_ftsn_set(sctp_ftsn_set_t **, sctp_faddr_t *, mblk_t *,
983                     uint_t *, uint32_t *);
984 extern void     sctp_add_recvq(sctp_t *, mblk_t *, boolean_t,
985                     ip_recv_attr_t *);
986 extern void     sctp_add_unrec_parm(sctp_parm_hdr_t *, mblk_t **, boolean_t);
987 extern size_t   sctp_addr_params(sctp_t *, int, uchar_t *, boolean_t);
988 extern mblk_t   *sctp_add_proto_hdr(sctp_t *, sctp_faddr_t *, mblk_t *, int,
989                     int *);
990 extern void     sctp_addr_req(sctp_t *, mblk_t *);
991 extern sctp_t   *sctp_addrlist2sctp(mblk_t *, sctp_hdr_t *, sctp_chunk_hdr_t *,
992                     zoneid_t, sctp_stack_t *);
993 extern void     sctp_check_adv_ack_pt(sctp_t *, mblk_t *, mblk_t *);
994 extern void     sctp_assoc_event(sctp_t *, uint16_t, uint16_t,
995                     sctp_chunk_hdr_t *);

997 extern void     sctp_bind_hash_insert(sctp_tf_t *, sctp_t *, int);
998 extern void     sctp_bind_hash_remove(sctp_t *);
999 extern int      sctp_bindi(sctp_t *, in_port_t, boolean_t, int, in_port_t *);
1000 extern int      sctp_bind_add(sctp_t *, const void *, uint32_t, boolean_t,
1001                     in_port_t);
1002 extern int      sctp_bind_del(sctp_t *, const void *, uint32_t, boolean_t);
1003 extern int      sctp_build_hdrs(sctp_t *, int);

1005 extern int      sctp_check_abandoned_msg(sctp_t *, mblk_t *);
1006 extern void     sctp_clean_death(sctp_t *, int);
1007 extern void     sctp_close_eager(sctp_t *);
1008 extern int      sctp_compare_faddrsets(sctp_faddr_t *, sctp_faddr_t *);
1009 extern void     sctp_congest_reset(sctp_t *);
1010 extern void     sctp_conn_hash_insert(sctp_tf_t *, sctp_t *, int);
1011 extern void     sctp_conn_hash_remove(sctp_t *);
1012 extern void     sctp_conn_init(conn_t *);
1013 extern sctp_t   *sctp_conn_match(in6_addr_t **, uint32_t, in6_addr_t *,
1014                     uint32_t, zoneid_t, iaflags_t, sctp_stack_t *);
```



```
1015 extern void     sctp_conn_reclaim(void *);
1016 extern sctp_t   *sctp_conn_request(sctp_t *, mblk_t *, uint_t, uint_t,
1017                     sctp_init_chunk_t *, ip_recv_attr_t *);
1018 extern uint32_t sctp_cumack(sctp_t *, uint32_t, mblk_t **);
1019 extern sctp_t   *sctp_create_eager(sctp_t *);

1021 extern void     sctp_dispatch_rput(queue_t *, sctp_t *, sctp_hdr_t *, mblk_t *,
1022                     uint_t, uint_t, in6_addr_t);
1023 extern char     *sctp_display(sctp_t *, char *);
1024 extern void     sctp_display_all(sctp_stack_t *);

1026 extern void     sctp_error_event(sctp_t *, sctp_chunk_hdr_t *, boolean_t);

1028 extern void     sctp_faddr_alive(sctp_t *, sctp_faddr_t *);
1029 extern int      sctp_faddr_dead(sctp_t *, sctp_faddr_t *, int);
1030 extern void     sctp_faddr_fini(void);
1031 extern void     sctp_faddr_init(void);
1032 extern void     sctp_fast_rexmit(sctp_t *);
1033 extern void     sctp_fill_sack(sctp_t *, unsigned char *, int);
1034 extern uint32_t sctp_find_listener_conf(sctp_stack_t *, in_port_t);
1035 extern void     sctp_free_faddr_timers(sctp_t *);
1036 extern void     sctp_free_ftsn_set(sctp_ftsn_set_t *);
1037 extern void     sctp_free_msg(mblk_t *);
1038 extern void     sctp_free_reass(sctp_instr_t *);
1039 extern void     sctp_free_set(sctp_set_t *);
1040 extern void     sctp_ftsn_sets_fini(void);
1041 extern void     sctp_ftsn_sets_init(void);

1043 extern int      sctp_get_addrlist(sctp_t *, const void *, uint32_t *,
1044                     uchar_t **, int *, size_t *);
1045 extern int      sctp_get_addrparams(sctp_t *, sctp_t *, mblk_t *,
1046                     sctp_chunk_hdr_t *, uint_t *);
1047 extern void     sctp_get_dest(sctp_t *, sctp_faddr_t *);
1048 extern void     sctp_get_faddr_list(sctp_t *, uchar_t *, size_t);
1049 extern mblk_t   *sctp_get_first_sent(sctp_t *);
1050 extern mblk_t   *sctp_get_msg_to_send(sctp_t *, mblk_t **, mblk_t *, int  *,
1051                     int32_t, uint32_t, sctp_faddr_t *);
1052 extern void     sctp_get_saddr_list(sctp_t *, uchar_t *, size_t);

1054 extern int      sctp_handle_error(sctp_t *, sctp_hdr_t *, sctp_chunk_hdr_t *,
1055                     mblk_t *, ip_recv_attr_t *);
1056 extern void     sctp_hash_destroy(sctp_stack_t *);
1057 extern void     sctp_hash_init(sctp_stack_t *);
1058 extern void     sctp_heartbeat_timer(sctp_t *);

1060 extern void     sctp_icmp_error(sctp_t *, mblk_t *);
1061 extern void     sctp_inc_taskq(sctp_stack_t *);
1062 extern void     sctp_info_req(sctp_t *, mblk_t *);
1063 extern mblk_t   *sctp_init_mp(sctp_t *, sctp_faddr_t *);
1064 extern boolean_t sctp_initialize_params(sctp_t *, sctp_init_chunk_t *,
1065                     sctp_init_chunk_t *);
1066 extern uint32_t sctp_init2vtag(sctp_chunk_hdr_t *);
1067 extern void     sctp_intf_event(sctp_t *, in6_addr_t, int, int);
1068 extern void     sctp_input_data(sctp_t *, mblk_t *, ip_recv_attr_t *);
1069 extern void     sctp_instream_cleanup(sctp_t *, boolean_t);
1070 extern boolean_t sctp_is_a_faddr_clean(sctp_t *);

1072 extern void     *sctp_kstat_init(netstackid_t);
1073 extern void     sctp_kstat_fini(netstackid_t, kstat_t *);
1074 extern void     *sctp_kstat2_init(netstackid_t);
1075 extern void     sctp_kstat2_fini(netstackid_t, kstat_t *);

1077 extern ssize_t  sctp_link_abort(mblk_t *, uint16_t, char *, size_t, int,
1078                     boolean_t);
1079 extern void     sctp_listen_hash_insert(sctp_tf_t *, sctp_t *);
1080 extern void     sctp_listen_hash_remove(sctp_t *);
```

```
1081 extern void     sctp_listener_conf_cleanup(sctp_stack_t *);
1082 extern sctp_t   *sctp_lookup(sctp_t *, in6_addr_t *, sctp_tf_t *, uint32_t *,
1083                     int);
1084 extern sctp_faddr_t *sctp_lookup_faddr(sctp_t *, in6_addr_t *);

1086 extern mblk_t   *sctp_make_err(sctp_t *, uint16_t, void *, size_t);
1087 extern mblk_t   *sctp_make_ftsn_chunk(sctp_t *, sctp_faddr_t *,
1088                     sctp_ftsn_set_t *, uint_t, uint32_t);
1089 extern void     sctp_make_ftsns(sctp_t *, mblk_t *, mblk_t *, mblk_t **,
1090                     sctp_faddr_t *, uint32_t *);
1091 extern mblk_t   *sctp_make_mp(sctp_t *, sctp_faddr_t *, int);
1092 extern mblk_t   *sctp_make_sack(sctp_t *, sctp_faddr_t *, mblk_t *);
1093 extern void     sctp_maxpsz_set(sctp_t *);
1094 extern void     sctp_move_faddr_timers(queue_t *, sctp_t *);

1096 extern sctp_parm_hdr_t *sctp_next_parm(sctp_parm_hdr_t *, ssize_t *);

1098 extern void     sctp_ootb_shutdown_ack(mblk_t *, uint_t, ip_recv_attr_t *,
1099                     ip_stack_t *);
1100 extern size_t   sctp_options_param(const sctp_t *, void *, int);
1101 extern size_t   sctp_options_param_len(const sctp_t *, int);
1102 extern void     sctp_output(sctp_t *, uint_t);

1104 extern void     sctp_partial_delivery_event(sctp_t *);
1105 extern int      sctp_process_cookie(sctp_t *, sctp_chunk_hdr_t *, mblk_t *,
1106                     sctp_init_chunk_t **, sctp_hdr_t *, int *, in6_addr_t *,
1107                     ip_recv_attr_t *);
1108 extern void     sctp_process_err(sctp_t *);
1109 extern void     sctp_process_heartbeat(sctp_t *, sctp_chunk_hdr_t *);
1110 extern void     sctp_process_timer(sctp_t *);

1112 extern void     sctp_redo_faddr_srcs(sctp_t *);
1113 extern void     sctp_regift_xmitlist(sctp_t *);
1114 extern void     sctp_return_heartbeat(sctp_t *, sctp_chunk_hdr_t *, mblk_t *);
1115 extern void     sctp_rexmit(sctp_t *, sctp_faddr_t *);
1116 extern mblk_t   *sctp_rexmit_packet(sctp_t *, mblk_t **, mblk_t **,
1117                     sctp_faddr_t *, uint_t *);
1118 extern void     sctp_rexmit_timer(sctp_t *, sctp_faddr_t *);
1119 extern sctp_faddr_t *sctp_rotate_faddr(sctp_t *, sctp_faddr_t *);

1121 extern boolean_t sctp_sack(sctp_t *, mblk_t *);
1122 extern int      sctp_secure_restart_check(mblk_t *, sctp_chunk_hdr_t *,
1123                     uint32_t, int, sctp_stack_t *, ip_recv_attr_t *);
1124 extern void     sctp_send_abort(sctp_t *, uint32_t, uint16_t, char *, size_t,
1125                     mblk_t *, int, boolean_t, ip_recv_attr_t *);
1126 extern void     sctp_ootb_send_abort(uint32_t, uint16_t, char *, size_t,
1127                     const mblk_t *, int, boolean_t, ip_recv_attr_t *,
1128                     ip_stack_t *);
1129 extern void     sctp_send_cookie_ack(sctp_t *);
1130 extern void     sctp_send_cookie_echo(sctp_t *, sctp_chunk_hdr_t *, mblk_t *,
1131                         ip_recv_attr_t *);
1132 extern void     sctp_send_initack(sctp_t *, sctp_hdr_t *, sctp_chunk_hdr_t *,
1133                     mblk_t *, ip_recv_attr_t *);
1134 extern void     sctp_send_shutdown(sctp_t *, int);
1135 extern void     sctp_send_heartbeat(sctp_t *, sctp_faddr_t *);
1136 extern void     sctp_sendfail_event(sctp_t *, mblk_t *, int, boolean_t);
1137 extern void     sctp_set_faddr_current(sctp_t *, sctp_faddr_t *);
1138 extern int      sctp_set_hdraddrs(sctp_t *);
1139 extern void     sctp_set_saddr(sctp_t *, sctp_faddr_t *);
1140 extern void     sctp_sets_init(void);
1141 extern void     sctp_sets_fini(void);
1142 extern void     sctp_shutdown_event(sctp_t *);
1143 extern void     sctp_stop_faddr_timers(sctp_t *);
1144 extern int      sctp_shutdown_received(sctp_t *, sctp_chunk_hdr_t *, boolean_t,
1145                     boolean_t, sctp_faddr_t *);
1146 extern void     sctp_shutdown_complete(sctp_t *);
```



```
1147 extern void     sctp_set_if_mtu(sctp_t *);
1148 extern void     sctp_set_iplen(sctp_t *, mblk_t *, ip_xmit_attr_t *);
1149 extern void     sctp_set_ulp_prop(sctp_t *);
1150 extern void     sctp_ss_rexmit(sctp_t *);
1151 extern void     sctp_stack_cpu_add(sctp_stack_t *, processorid_t);
1152 extern size_t   sctp_supaddr_param_len(sctp_t *);
1153 extern size_t   sctp_supaddr_param(sctp_t *, uchar_t *);

1155 extern void     sctp_timer(sctp_t *, mblk_t *, clock_t);
1156 extern mblk_t   *sctp_timer_alloc(sctp_t *, pfv_t, int);
1157 extern void     sctp_timer_call(sctp_t *sctp, mblk_t *);
1158 extern void     sctp_timer_free(mblk_t *);
1159 extern void     sctp_timer_stop(mblk_t *);
1160 extern void     sctp_unlink_faddr(sctp_t *, sctp_faddr_t *);

1162 extern void     sctp_update_dce(sctp_t *sctp);
1163 extern in_port_t sctp_update_next_port(in_port_t, zone_t *zone, sctp_stack_t *);
1164 extern void     sctp_update_rtt(sctp_t *, sctp_faddr_t *, clock_t);
1165 extern void     sctp_user_abort(sctp_t *, mblk_t *);

1167 extern void     sctp_validate_peer(sctp_t *);

1169 extern int      sctp_xmit_list_clean(sctp_t *, ssize_t);

1171 extern void     sctp_zap_addrs(sctp_t *);
1172 extern void     sctp_zap_faddrs(sctp_t *, int);
1173 extern sctp_chunk_hdr_t *sctp_first_chunk(uchar_t *, ssize_t);
1174 extern void     sctp_send_shutdown_ack(sctp_t *, sctp_faddr_t *, boolean_t);

1176 /* Contract private interface between SCTP and Clustering - PSARC/2005/602 */

1178 extern void     (*cl_sctp_listen)(sa_family_t, uchar_t *, uint_t, in_port_t);
1179 extern void     (*cl_sctp_unlisten)(sa_family_t, uchar_t *, uint_t, in_port_t);
1180 extern void     (*cl_sctp_connect)(sa_family_t, uchar_t *, uint_t, in_port_t,
1181                     uchar_t *, uint_t, in_port_t, boolean_t, cl_sctp_handle_t);
1182 extern void     (*cl_sctp_disconnect)(sa_family_t, cl_sctp_handle_t);
1183 extern void     (*cl_sctp_assoc_change)(sa_family_t, uchar_t *, size_t, uint_t,
1184                     uchar_t *, size_t, uint_t, int, cl_sctp_handle_t);
1185 extern void     (*cl_sctp_check_addrs)(sa_family_t, in_port_t, uchar_t **,
1186                     size_t, uint_t *, boolean_t);

1188 #define RUN_SCTP(sctp)                                          \
1189 {                                                               \
1190         mutex_enter(&(sctp)->sctp_lock);                       \
1191         while ((sctp)->sctp_running)                           \
1192                 cv_wait(&(sctp)->sctp_cv, &(sctp)->sctp_lock);  \
1193         (sctp)->sctp_running = B_TRUE;                         \
1194         mutex_exit(&(sctp)->sctp_lock);                        \
1195 }

1197 /* Wake up recvq taskq */
1198 #define WAKE_SCTP(sctp)                                 \
1199 {                                                       \
1200         mutex_enter(&(sctp)->sctp_lock);               \
1201         if ((sctp)->sctp_timer_mp != NULL)             \
1202                 sctp_process_timer(sctp);              \
1203         (sctp)->sctp_running = B_FALSE;                \
1204         cv_broadcast(&(sctp)->sctp_cv);                \
1205         mutex_exit(&(sctp)->sctp_lock);                \
1206 }

1208 #ifdef  __cplusplus
1209 }
1210 #endif

1212 #endif  /* _INET_SCTP_SCTP_IMPL_H */
```

```
**********************************************************
  129612 Mon Jul  9 14:38:20 2012
new/usr/src/uts/common/inet/tcp/tcp.c
tcp: conn_mlp_type will be set later in tcp_init_values
**********************************************************
______unchanged_portion_omitted_

2594 conn_t *
2595 tcp_create_common(cred_t *credp, boolean_t isv6, boolean_t issocket,
2596     int *errorp)
2597 {
2598         tcp_t           *tcp = NULL;
2599         conn_t          *connp;
2600         zoneid_t        zoneid;
2601         tcp_stack_t     *tcps;
2602         squeue_t        *sqp;

2604         ASSERT(errorp != NULL);
2605         /*
2606          * Find the proper zoneid and netstack.
2607          */
2608         /*
2609          * Special case for install: miniroot needs to be able to
2610          * access files via NFS as though it were always in the
2611          * global zone.
2612          */
2613         if (credp == kcred && nfs_global_client_only != 0) {
2614                 zoneid = GLOBAL_ZONEID;
2615                 tcps = netstack_find_by_stackid(GLOBAL_NETSTACKID)->
2616                     netstack_tcp;
2617                 ASSERT(tcps != NULL);
2618         } else {
2619                 netstack_t *ns;
2620                 int err;

2622                 if ((err = secpolicy_basic_net_access(credp)) != 0) {
2623                         *errorp = err;
2624                         return (NULL);
2625                 }

2627                 ns = netstack_find_by_cred(credp);
2628                 ASSERT(ns != NULL);
2629                 tcps = ns->netstack_tcp;
2630                 ASSERT(tcps != NULL);

2632                 /*
2633                  * For exclusive stacks we set the zoneid to zero
2634                  * to make TCP operate as if in the global zone.
2635                  */
2636                 if (tcps->tcps_netstack->netstack_stackid !=
2637                     GLOBAL_NETSTACKID)
2638                         zoneid = GLOBAL_ZONEID;
2639                 else
2640                         zoneid = crgetzoneid(credp);
2641         }

2643         sqp = IP_SQUEUE_GET((uint_t)gethrtime());
2644         connp = (conn_t *)tcp_get_conn(sqp, tcps);
2645         /*
2646          * Both tcp_get_conn and netstack_find_by_cred incremented refcnt,
2647          * so we drop it by one.
2648          */
2649         netstack_rele(tcps->tcps_netstack);
2650         if (connp == NULL) {
2651                 *errorp = ENOSR;
2652                 return (NULL);
```



```
2653         }
2654         ASSERT(connp->conn_ixa->ixa_protocol == connp->conn_proto);

2656         connp->conn_sqp = sqp;
2657         connp->conn_initial_sqp = connp->conn_sqp;
2658         connp->conn_ixa->ixa_sqp = connp->conn_sqp;
2659         tcp = connp->conn_tcp;

2661         /*
2662          * Besides asking IP to set the checksum for us, have conn_ip_output
2663          * to do the following checks when necessary:
2664          *
2665          * IXAF_VERIFY_SOURCE: drop packets when our outer source goes invalid
2666          * IXAF_VERIFY_PMTU: verify PMTU changes
2667          * IXAF_VERIFY_LSO: verify LSO capability changes
2668          */
2669         connp->conn_ixa->ixa_flags |= IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
2670             IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO;

2672         if (!tcps->tcps_dev_flow_ctl)
2673                 connp->conn_ixa->ixa_flags |= IXAF_NO_DEV_FLOW_CTL;

2675         if (isv6) {
2676                 connp->conn_ixa->ixa_src_preferences = IPV6_PREFER_SRC_DEFAULT;
2677                 connp->conn_ipversion = IPV6_VERSION;
2678                 connp->conn_family = AF_INET6;
2679                 tcp->tcp_mss = tcps->tcps_mss_def_ipv6;
2680                 connp->conn_default_ttl = tcps->tcps_ipv6_hoplimit;
2681         } else {
2682                 connp->conn_ipversion = IPV4_VERSION;
2683                 connp->conn_family = AF_INET;
2684                 tcp->tcp_mss = tcps->tcps_mss_def_ipv4;
2685                 connp->conn_default_ttl = tcps->tcps_ipv4_ttl;
2686         }
2687         connp->conn_xmit_ipp.ipp_unicast_hops = connp->conn_default_ttl;

2689         crhold(credp);
2690         connp->conn_cred = credp;
2691         connp->conn_cpid = curproc->p_pid;
2692         connp->conn_open_time = ddi_get_lbolt64();

2694         /* Cache things in the ixa without any refhold */
2695         ASSERT(!(connp->conn_ixa->ixa_free_flags & IXA_FREE_CRED));
2696         connp->conn_ixa->ixa_cred = credp;
2697         connp->conn_ixa->ixa_cpid = connp->conn_cpid;

2699         connp->conn_zoneid = zoneid;
2700         /* conn_allzones can not be set this early, hence no IPCL_ZONEID */
2701         connp->conn_ixa->ixa_zoneid = zoneid;
2702         connp->conn_mlp_type = mlptSingle;
2702         ASSERT(connp->conn_netstack == tcps->tcps_netstack);
2703         ASSERT(tcp->tcp_tcps == tcps);

2705         /*
2706          * If the caller has the process-wide flag set, then default to MAC
2707          * exempt mode.  This allows read-down to unlabeled hosts.
2708          */
2709         if (getpflags(NET_MAC_AWARE, credp) != 0)
2710                 connp->conn_mac_mode = CONN_MAC_AWARE;

2712         connp->conn_zone_is_global = (crgetzoneid(credp) == GLOBAL_ZONEID);

2714         if (issocket) {
2715                 tcp->tcp_issocket = 1;
2716         }
```

```
2718         connp->conn_rcvbuf = tcps->tcps_recv_hiwat;
2719         connp->conn_sndbuf = tcps->tcps_xmit_hiwat;
2720         connp->conn_sndlowat = tcps->tcps_xmit_lowat;
2721         connp->conn_so_type = SOCK_STREAM;
2722         connp->conn_wroff = connp->conn_ht_iphc_allocated +
2723             tcps->tcps_wroff_xtra;

2725         SOCK_CONNID_INIT(tcp->tcp_connid);
2726         /* DTrace ignores this - it isn't a tcp:::state-change */
2727         tcp->tcp_state = TCPS_IDLE;
2728         tcp_init_values(tcp, NULL);
2729         return (connp);
2730 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30240 Mon Jul  9 14:38:20 2012
new/usr/src/uts/common/inet/tcp/tcp_opt_data.c
tcp: maybe related to 721fffe3
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 Nexenta Systems, Inc. All rights reserved.
 */

#include <sys/types.h>
#include <sys/stream.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/socket.h>
#include <sys/xti_xtiopt.h>
#include <sys/xti_inet.h>
#include <sys/policy.h>

#include <inet/common.h>
#include <netinet/ip6.h>
#include <inet/ip.h>

#include <netinet/in.h>
#include <netinet/tcp.h>
#include <inet/optcom.h>
#include <inet/proto_set.h>
#include <inet/tcp_impl.h>

static int      tcp_opt_default(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);

#endif /* ! codereview */
/*
 * Table of all known options handled on a TCP protocol stack.
 *
 * Note: This table contains options processed by both TCP and IP levels
 *       and is the superset of options that can be performed on a TCP over IP
 *       stack.
 */
opdes_t tcp_opt_arr[] = {

{ SO_LINGER,    SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0,
        sizeof (struct linger), 0 },

{ SO_DEBUG,     SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_KEEPALIVE, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
```



```
{ SO_DONTROUTE, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_USELOOPBACK, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0
        },
{ SO_BROADCAST, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_REUSEADDR, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_OOBINLINE, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_TYPE,      SOL_SOCKET, OA_R, OA_R, OP_NP, 0, sizeof (int), 0 },
{ SO_SNDBUF,    SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_RCVBUF,    SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_SNDTIMEO,  SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0,
        sizeof (struct timeval), 0 },
{ SO_RCVTIMEO,  SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0,
        sizeof (struct timeval), 0 },
{ SO_DGRAM_ERRIND, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0
        },
{ SO_SND_COPYAVOID, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ SO_ANON_MLP, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
        0 },
{ SO_MAC_EXEMPT, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
        0 },
{ SO_MAC_IMPLICIT, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
        0 },
{ SO_ALLZONES, SOL_SOCKET, OA_R, OA_RW, OP_CONFIG, 0, sizeof (int),
        0 },
{ SO_EXCLBIND, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

{ SO_DOMAIN,    SOL_SOCKET, OA_R, OA_R, OP_NP, 0, sizeof (int), 0 },

{ SO_PROTOTYPE, SOL_SOCKET, OA_R, OA_R, OP_NP, 0, sizeof (int), 0 },

{ TCP_NODELAY,  IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0
        },
{ TCP_MAXSEG,   IPPROTO_TCP, OA_R, OA_R, OP_NP, 0, sizeof (uint_t),
        536 },

{ TCP_NOTIFY_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
        OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

{ TCP_ABORT_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
        OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

{ TCP_CONN_NOTIFY_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
        OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

{ TCP_CONN_ABORT_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
        OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

{ TCP_RECVDSTADDR, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
        0 },

{ TCP_ANONPRIVBIND, IPPROTO_TCP, OA_R, OA_RW, OP_PRIVPORT, 0,
        sizeof (int), 0 },

{ TCP_EXCLBIND, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0
        },

{ TCP_INIT_CWND, IPPROTO_TCP, OA_RW, OA_RW, OP_CONFIG, 0,
        sizeof (int), 0 },

{ TCP_KEEPALIVE_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },

{ TCP_KEEPIDLE, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

{ TCP_KEEPCNT, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
```

```
{ TCP_KEEPINTVL, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

{ TCP_KEEPALIVE_ABORT_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },

{ TCP_CORK, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

{ TCP_RTO_INITIAL, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (uint32_t), 0 },

{ TCP_RTO_MIN, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (uint32_t), 0 },

{ TCP_RTO_MAX, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (uint32_t), 0 },

{ TCP_LINGER2, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

{ IP_OPTIONS,   IPPROTO_IP, OA_RW, OA_RW, OP_NP,
        (OP_VARLEN|OP_NODEFAULT),
        IP_MAX_OPT_LENGTH + IP_ADDR_LEN, -1 /* not initialized */ },
{ T_IP_OPTIONS, IPPROTO_IP, OA_RW, OA_RW, OP_NP,
        (OP_VARLEN|OP_NODEFAULT),
        IP_MAX_OPT_LENGTH + IP_ADDR_LEN, -1 /* not initialized */ },

{ IP_TOS,       IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ T_IP_TOS,     IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
{ IP_TTL,       IPPROTO_IP, OA_RW, OA_RW, OP_NP, OP_DEF_FN,
        sizeof (int), -1 /* not initialized */ },

{ IP_SEC_OPT, IPPROTO_IP, OA_RW, OA_RW, OP_NP, OP_NODEFAULT,
        sizeof (ipsec_req_t), -1 /* not initialized */ },

{ IP_BOUND_IF, IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int),   0 /* no ifindex */ },

{ IP_UNSPEC_SRC, IPPROTO_IP, OA_R, OA_RW, OP_RAW, 0,
        sizeof (int), 0 },

{ IPV6_UNICAST_HOPS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, OP_DEF_FN,
        sizeof (int), -1 /* not initialized */ },

{ IPV6_BOUND_IF, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int),   0 /* no ifindex */ },

{ IP_DONTFRAG, IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

{ IP_NEXTHOP, IPPROTO_IP, OA_R, OA_RW, OP_CONFIG, 0,
        sizeof (in_addr_t),     -1 /* not initialized  */ },

{ IPV6_UNSPEC_SRC, IPPROTO_IPV6, OA_R, OA_RW, OP_RAW, 0,
        sizeof (int), 0 },

{ IPV6_PKTINFO, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
        (OP_NODEFAULT|OP_VARLEN),
        sizeof (struct in6_pktinfo), -1 /* not initialized */ },
{ IPV6_NEXTHOP, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
        OP_NODEFAULT,
        sizeof (sin6_t), -1 /* not initialized */ },
{ IPV6_HOPOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
        (OP_VARLEN|OP_NODEFAULT), 255*8,
        -1 /* not initialized */ },
{ IPV6_DSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
        (OP_VARLEN|OP_NODEFAULT), 255*8,
        -1 /* not initialized */ },
{ IPV6_RTHDRDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
        (OP_VARLEN|OP_NODEFAULT), 255*8,
        -1 /* not initialized */ },
{ IPV6_RTHDR, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
```



```
        (OP_VARLEN|OP_NODEFAULT), 255*8,
        -1 /* not initialized */ },
{ IPV6_TCLASS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
        OP_NODEFAULT,
        sizeof (int), -1 /* not initialized */ },
{ IPV6_PATHMTU, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
        OP_NODEFAULT,
        sizeof (struct ip6_mtuinfo), -1 /* not initialized */ },
{ IPV6_DONTFRAG, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_USE_MIN_MTU, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_V6ONLY, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },

/* Enable receipt of ancillary data */
{ IPV6_RECVPKTINFO, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_RECVHOPLIMIT, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_RECVHOPOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ _OLD_IPV6_RECVDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_RECVDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_RECVRTHDR, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_RECVRTHDRDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },
{ IPV6_RECVTCLASS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (int), 0 },

{ IPV6_SEC_OPT, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, OP_NODEFAULT,
        sizeof (ipsec_req_t), -1 /* not initialized */ },
{ IPV6_SRC_PREFERENCES, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
        sizeof (uint32_t), IPV6_PREFER_SRC_DEFAULT },
};

/*
 * Table of all supported levels
 * Note: Some levels (e.g. XTI_GENERIC) may be valid but may not have
 * any supported options so we need this info separately.
 *
 * This is needed only for topmost tpi providers and is used only by
 * XTI interfaces.
 */
optlevel_t      tcp_valid_levels_arr[] = {
        XTI_GENERIC,
        SOL_SOCKET,
        IPPROTO_TCP,
        IPPROTO_IP,
        IPPROTO_IPV6
};


#define TCP_OPT_ARR_CNT         A_CNT(tcp_opt_arr)
#define TCP_VALID_LEVELS_CNT    A_CNT(tcp_valid_levels_arr)

uint_t tcp_max_optsize; /* initialized when TCP driver is loaded */

/*
 * Initialize option database object for TCP
 *
 * This object represents database of options to search passed to
 * {sock,tpi}optcom_req() interface routine to take care of option
```

```
 * management and associated methods.
 */

optdb_obj_t tcp_opt_obj = {
        tcp_opt_default,        /* TCP default value function pointer */
        tcp_tpi_opt_get,        /* TCP get function pointer */
        tcp_tpi_opt_set,        /* TCP set function pointer */
        TCP_OPT_ARR_CNT,        /* TCP option database count of entries */
        tcp_opt_arr,            /* TCP option database */
        TCP_VALID_LEVELS_CNT,   /* TCP valid level count of entries */
        tcp_valid_levels_arr    /* TCP valid level array */
};

/* Maximum TCP initial cwin (start/restart). */
#define TCP_MAX_INIT_CWND       16

static int tcp_max_init_cwnd = TCP_MAX_INIT_CWND;

/*
 * Some TCP options can be "set" by requesting them in the option
 * buffer. This is needed for XTI feature test though we do not
 * allow it in general. We interpret that this mechanism is more
 * applicable to OSI protocols and need not be allowed in general.
 * This routine filters out options for which it is not allowed (most)
 * and lets through those (few) for which it is. [ The XTI interface
 * test suite specifics will imply that any XTI_GENERIC level XTI_* if
 * ever implemented will have to be allowed here ].
 */
static boolean_t
tcp_allow_connopt_set(int level, int name)
{

        switch (level) {
        case IPPROTO_TCP:
                switch (name) {
                case TCP_NODELAY:
                        return (B_TRUE);
                default:
                        return (B_FALSE);
                }
                /*NOTREACHED*/
        default:
                return (B_FALSE);
        }
        /*NOTREACHED*/
}

/*
 * This routine gets default values of certain options whose default
 * values are maintained by protocol specific code
 */
/* ARGSUSED */
int
tcp_opt_default(queue_t *q, int level, int name, uchar_t *ptr)
{
        int32_t *i1 = (int32_t *)ptr;
        tcp_stack_t     *tcps = Q_TO_TCP(q)->tcp_tcps;

        switch (level) {
        case IPPROTO_TCP:
                switch (name) {
                case TCP_NOTIFY_THRESHOLD:
                        *i1 = tcps->tcps_ip_notify_interval;
                        break;
                case TCP_ABORT_THRESHOLD:
                        *i1 = tcps->tcps_ip_abort_interval;
```



```
                        break;
                case TCP_CONN_NOTIFY_THRESHOLD:
                        *i1 = tcps->tcps_ip_notify_cinterval;
                        break;
                case TCP_CONN_ABORT_THRESHOLD:
                        *i1 = tcps->tcps_ip_abort_cinterval;
                        break;
                default:
                        return (-1);
                }
                break;
        case IPPROTO_IP:
                switch (name) {
                case IP_TTL:
                        *i1 = tcps->tcps_ipv4_ttl;
                        break;
                default:
                        return (-1);
                }
                break;
        case IPPROTO_IPV6:
                switch (name) {
                case IPV6_UNICAST_HOPS:
                        *i1 = tcps->tcps_ipv6_hoplimit;
                        break;
                default:
                        return (-1);
                }
                break;
        default:
                return (-1);
        }
        return (sizeof (int));
}

/*
 * TCP routine to get the values of options.
 */
int
tcp_opt_get(conn_t *connp, int level, int name, uchar_t *ptr)
{
        int             *i1 = (int *)ptr;
        tcp_t           *tcp = connp->conn_tcp;
        conn_opt_arg_t  coas;
        int             retval;

        coas.coa_connp = connp;
        coas.coa_ixa = connp->conn_ixa;
        coas.coa_ipp = &connp->conn_xmit_ipp;
        coas.coa_ancillary = B_FALSE;
        coas.coa_changed = 0;

        switch (level) {
        case SOL_SOCKET:
                switch (name) {
                case SO_SND_COPYAVOID:
                        *i1 = tcp->tcp_snd_zcopy_on ?
                            SO_SND_COPYAVOID : 0;
                        return (sizeof (int));
                case SO_ACCEPTCONN:
                        *i1 = (tcp->tcp_state == TCPS_LISTEN);
                        return (sizeof (int));
                }
                break;
        case IPPROTO_TCP:
                switch (name) {
```

```
392                 case TCP_NODELAY:
393                         *i1 = (tcp->tcp_naglim == 1) ? TCP_NODELAY : 0;
394                         return (sizeof (int));
395                 case TCP_MAXSEG:
396                         *i1 = tcp->tcp_mss;
397                         return (sizeof (int));
398                 case TCP_NOTIFY_THRESHOLD:
399                         *i1 = (int)tcp->tcp_first_timer_threshold;
400                         return (sizeof (int));
401                 case TCP_ABORT_THRESHOLD:
402                         *i1 = tcp->tcp_second_timer_threshold;
403                         return (sizeof (int));
404                 case TCP_CONN_NOTIFY_THRESHOLD:
405                         *i1 = tcp->tcp_first_ctimer_threshold;
406                         return (sizeof (int));
407                 case TCP_CONN_ABORT_THRESHOLD:
408                         *i1 = tcp->tcp_second_ctimer_threshold;
409                         return (sizeof (int));
410                 case TCP_INIT_CWND:
411                         *i1 = tcp->tcp_init_cwnd;
412                         return (sizeof (int));
413                 case TCP_KEEPALIVE_THRESHOLD:
414                         *i1 = tcp->tcp_ka_interval;
415                         return (sizeof (int));

417                 /*
418                  * TCP_KEEPIDLE expects value in seconds, but
419                  * tcp_ka_interval is in milliseconds.
420                  */
421                 case TCP_KEEPIDLE:
422                         *i1 = tcp->tcp_ka_interval / 1000;
423                         return (sizeof (int));
424                 case TCP_KEEPCNT:
425                         *i1 = tcp->tcp_ka_cnt;
426                         return (sizeof (int));

428                 /*
429                  * TCP_KEEPINTVL expects value in seconds, but
430                  * tcp_ka_rinterval is in milliseconds.
431                  */
432                 case TCP_KEEPINTVL:
433                         *i1 = tcp->tcp_ka_rinterval / 1000;
434                         return (sizeof (int));
435                 case TCP_KEEPALIVE_ABORT_THRESHOLD:
436                         *i1 = tcp->tcp_ka_abort_thres;
437                         return (sizeof (int));
438                 case TCP_CORK:
439                         *i1 = tcp->tcp_cork;
440                         return (sizeof (int));
441                 case TCP_RTO_INITIAL:
442                         *i1 = tcp->tcp_rto_initial;
443                         return (sizeof (uint32_t));
444                 case TCP_RTO_MIN:
445                         *i1 = tcp->tcp_rto_min;
446                         return (sizeof (uint32_t));
447                 case TCP_RTO_MAX:
448                         *i1 = tcp->tcp_rto_max;
449                         return (sizeof (uint32_t));
450                 case TCP_LINGER2:
451                         *i1 = tcp->tcp_fin_wait_2_flush_interval / SECONDS;
452                         return (sizeof (int));
453                 }
454                 break;
455         case IPPROTO_IP:
456                 if (connp->conn_family != AF_INET)
457                         return (-1);
```



```
458                 switch (name) {
459                 case IP_OPTIONS:
460                 case T_IP_OPTIONS:
461                         /* Caller ensures enough space */
462                         return (ip_opt_get_user(connp, ptr));
463                 default:
464                         break;
465                 }
466                 break;

468         case IPPROTO_IPV6:
469                 /*
470                  * IPPROTO_IPV6 options are only supported for sockets
471                  * that are using IPv6 on the wire.
472                  */
473                 if (connp->conn_ipversion != IPV6_VERSION) {
474                         return (-1);
475                 }
476                 switch (name) {
477                 case IPV6_PATHMTU:
478                         if (tcp->tcp_state < TCPS_ESTABLISHED)
479                                 return (-1);
480                         break;
481                 }
482                 break;
483         }
484         mutex_enter(&connp->conn_lock);
485         retval = conn_opt_get(&coas, level, name, ptr);
486         mutex_exit(&connp->conn_lock);
487         return (retval);
488 }

490 /*
491  * We declare as 'int' rather than 'void' to satisfy pfi_t arg requirements.
492  * Parameters are assumed to be verified by the caller.
493  */
494 /* ARGSUSED */
495 int
496 tcp_opt_set(conn_t *connp, uint_t optset_context, int level, int name,
497     uint_t inlen, uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
498     void *thisdg_attrs, cred_t *cr)
499 {
500         tcp_t   *tcp = connp->conn_tcp;
501         int     *i1 = (int *)invalp;
502         boolean_t onoff = (*i1 == 0) ? 0 : 1;
503         boolean_t checkonly;
504         int     reterr;
505         tcp_stack_t     *tcps = tcp->tcp_tcps;
506         conn_opt_arg_t  coas;
507         uint32_t        val = *((uint32_t *)invalp);

509         coas.coa_connp = connp;
510         coas.coa_ixa = connp->conn_ixa;
511         coas.coa_ipp = &connp->conn_xmit_ipp;
512         coas.coa_ancillary = B_FALSE;
513         coas.coa_changed = 0;

515         switch (optset_context) {
516         case SETFN_OPTCOM_CHECKONLY:
517                 checkonly = B_TRUE;
518                 /*
519                  * Note: Implies T_CHECK semantics for T_OPTCOM_REQ
520                  * inlen != 0 implies value supplied and
521                  *      we have to "pretend" to set it.
522                  * inlen == 0 implies that there is no
523                  *      value part in T_CHECK request and just validation
```

```
524                  * done elsewhere should be enough, we just return here.
525                  */
526                 if (inlen == 0) {
527                         *outlenp = 0;
528                         return (0);
529                 }
530                 break;
531         case SETFN_OPTCOM_NEGOTIATE:
532                 checkonly = B_FALSE;
533                 break;
534         case SETFN_UD_NEGOTIATE: /* error on conn-oriented transports ? */
535         case SETFN_CONN_NEGOTIATE:
536                 checkonly = B_FALSE;
537                 /*
538                  * Negotiating local and "association-related" options
539                  * from other (T_CONN_REQ, T_CONN_RES,T_UNITDATA_REQ)
540                  * primitives is allowed by XTI, but we choose
541                  * to not implement this style negotiation for Internet
542                  * protocols (We interpret it is a must for OSI world but
543                  * optional for Internet protocols) for all options.
544                  * [ Will do only for the few options that enable test
545                  * suites that our XTI implementation of this feature
546                  * works for transports that do allow it ]
547                  */
548                 if (!tcp_allow_connopt_set(level, name)) {
549                         *outlenp = 0;
550                         return (EINVAL);
551                 }
552                 break;
553         default:
554                 /*
555                  * We should never get here
556                  */
557                 *outlenp = 0;
558                 return (EINVAL);
559         }

561         ASSERT((optset_context != SETFN_OPTCOM_CHECKONLY) ||
562             (optset_context == SETFN_OPTCOM_CHECKONLY && inlen != 0));

564         /*
565          * For TCP, we should have no ancillary data sent down
566          * (sendmsg isn't supported for SOCK_STREAM), so thisdg_attrs
567          * has to be zero.
568          */
569         ASSERT(thisdg_attrs == NULL);

571         /*
572          * For fixed length options, no sanity check
573          * of passed in length is done. It is assumed *_optcom_req()
574          * routines do the right thing.
575          */
576         switch (level) {
577         case SOL_SOCKET:
578                 switch (name) {
579                 case SO_KEEPALIVE:
580                         if (checkonly) {
581                                 /* check only case */
582                                 break;
583                         }

585                         if (!onoff) {
586                                 if (connp->conn_keepalive) {
587                                         if (tcp->tcp_ka_tid != 0) {
588                                                 (void) TCP_TIMER_CANCEL(tcp,
589                                                     tcp->tcp_ka_tid);
```



```
590                                                 tcp->tcp_ka_tid = 0;
591                                         }
592                                         connp->conn_keepalive = 0;
593                                 }
594                                 break;
595                         }
596                         if (!connp->conn_keepalive) {
597                                 /* Crank up the keepalive timer */
598                                 tcp->tcp_ka_last_intrvl = 0;
599                                 tcp->tcp_ka_tid = TCP_TIMER(tcp,
600                                     tcp_keepalive_timer, tcp->tcp_ka_interval);
601                                 connp->conn_keepalive = 1;
602                         }
603                         break;
604                 case SO_SNDBUF: {
605                         if (*i1 > tcps->tcps_max_buf) {
606                                 *outlenp = 0;
607                                 return (ENOBUFS);
608                         }
609                         if (checkonly)
610                                 break;

612                         connp->conn_sndbuf = *i1;
613                         if (tcps->tcps_snd_lowat_fraction != 0) {
614                                 connp->conn_sndlowat = connp->conn_sndbuf /
615                                     tcps->tcps_snd_lowat_fraction;
616                         }
617                         (void) tcp_maxpsz_set(tcp, B_TRUE);
618                         /*
619                          * If we are flow-controlled, recheck the condition.
620                          * There are apps that increase SO_SNDBUF size when
621                          * flow-controlled (EWOULDBLOCK), and expect the flow
622                          * control condition to be lifted right away.
623                          */
624                         mutex_enter(&tcp->tcp_non_sq_lock);
625                         if (tcp->tcp_flow_stopped &&
626                             TCP_UNSENT_BYTES(tcp) < connp->conn_sndbuf) {
627                                 tcp_clrqfull(tcp);
628                         }
629                         mutex_exit(&tcp->tcp_non_sq_lock);
630                         *outlenp = inlen;
631                         return (0);
632                 }
633                 case SO_RCVBUF:
634                         if (*i1 > tcps->tcps_max_buf) {
635                                 *outlenp = 0;
636                                 return (ENOBUFS);
637                         }
638                         /* Silently ignore zero */
639                         if (!checkonly && *i1 != 0) {
640                                 *i1 = MSS_ROUNDUP(*i1, tcp->tcp_mss);
641                                 (void) tcp_rwnd_set(tcp, *i1);
642                         }
643                         /*
644                          * XXX should we return the rwnd here
645                          * and tcp_opt_get ?
646                          */
647                         *outlenp = inlen;
648                         return (0);
649                 case SO_SND_COPYAVOID:
650                         if (!checkonly) {
651                                 if (tcp->tcp_loopback ||
652                                     (onoff != 1) || !tcp_zcopy_check(tcp)) {
653                                         *outlenp = 0;
654                                         return (EOPNOTSUPP);
655                                 }
```

```
656                             tcp->tcp_snd_zcopy_aware = 1;
657                     }
658                     *outlenp = inlen;
659                     return (0);
660             }
661             break;
662     case IPPROTO_TCP:
663             switch (name) {
664             case TCP_NODELAY:
665                     if (!checkonly)
666                             tcp->tcp_naglim = *i1 ? 1 : tcp->tcp_mss;
667                     break;
668             case TCP_NOTIFY_THRESHOLD:
669                     if (!checkonly)
670                             tcp->tcp_first_timer_threshold = *i1;
671                     break;
672             case TCP_ABORT_THRESHOLD:
673                     if (!checkonly)
674                             tcp->tcp_second_timer_threshold = *i1;
675                     break;
676             case TCP_CONN_NOTIFY_THRESHOLD:
677                     if (!checkonly)
678                             tcp->tcp_first_ctimer_threshold = *i1;
679                     break;
680             case TCP_CONN_ABORT_THRESHOLD:
681                     if (!checkonly)
682                             tcp->tcp_second_ctimer_threshold = *i1;
683                     break;
684             case TCP_RECVDSTADDR:
685                     if (tcp->tcp_state > TCPS_LISTEN) {
686                             *outlenp = 0;
687                             return (EOPNOTSUPP);
688                     }
689                     /* Setting done in conn_opt_set */
690                     break;
691             case TCP_INIT_CWND:
692                     if (checkonly)
693                             break;

695                     /*
696                      * Only allow socket with network configuration
697                      * privilege to set the initial cwnd to be larger
698                      * than allowed by RFC 3390.
699                      */
700                     if (val > MIN(4, MAX(2, 4380 / tcp->tcp_mss))) {
701                             if ((reterr = secpolicy_ip_config(cr, B_TRUE))
702                                 != 0) {
703                                     *outlenp = 0;
704                                     return (reterr);
705                             }
706                             if (val > tcp_max_init_cwnd) {
707                                     *outlenp = 0;
708                                     return (EINVAL);
709                             }
710                     }

712                     tcp->tcp_init_cwnd = val;

714                     /*
715                      * If the socket is connected, AND no outbound data
716                      * has been sent, reset the actual cwnd values.
717                      */
718                     if (tcp->tcp_state == TCPS_ESTABLISHED &&
719                         tcp->tcp_iss == tcp->tcp_snxt - 1) {
720                             tcp->tcp_cwnd =
721                                 MIN(tcp->tcp_rwnd, val * tcp->tcp_mss);
```



```
722                     }
723                     break;

725             /*
726              * TCP_KEEPIDLE is in seconds but TCP_KEEPALIVE_THRESHOLD
727              * is in milliseconds. TCP_KEEPIDLE is introduced for
728              * compatibility with other Unix flavors.
729              * We can fall through TCP_KEEPALIVE_THRESHOLD logic after
730              * converting the input to milliseconds.
731              */
732             case TCP_KEEPIDLE:
733                     *i1 *= 1000;
734                     /* FALLTHRU */

736             case TCP_KEEPALIVE_THRESHOLD:
737                     if (checkonly)
738                             break;

740                     if (*i1 < tcps->tcps_keepalive_interval_low ||
741                         *i1 > tcps->tcps_keepalive_interval_high) {
742                             *outlenp = 0;
743                             return (EINVAL);
744                     }
745                     if (*i1 != tcp->tcp_ka_interval) {
746                             tcp->tcp_ka_interval = *i1;
747                             /*
748                              * Check if we need to restart the
749                              * keepalive timer.
750                              */
751                             if (tcp->tcp_ka_tid != 0) {
752                                     ASSERT(connp->conn_keepalive);
753                                     (void) TCP_TIMER_CANCEL(tcp,
754                                         tcp->tcp_ka_tid);
755                                     tcp->tcp_ka_last_intrvl = 0;
756                                     tcp->tcp_ka_tid = TCP_TIMER(tcp,
757                                         tcp_keepalive_timer,
758                                         tcp->tcp_ka_interval);
759                             }
760                     }
761                     break;

763             /*
764              * tcp_ka_abort_thres = tcp_ka_rinterval * tcp_ka_cnt.
765              * So setting TCP_KEEPCNT or TCP_KEEPINTVL can affect all the
766              * three members - tcp_ka_abort_thres, tcp_ka_rinterval and
767              * tcp_ka_cnt.
768              */
769             case TCP_KEEPCNT:
770                     if (checkonly)
771                             break;

773                     if (*i1 == 0) {
774                             return (EINVAL);
775                     } else if (tcp->tcp_ka_rinterval == 0) {
776                             if ((tcp->tcp_ka_abort_thres / *i1) <
777                                 tcp->tcp_rto_min ||
778                                 (tcp->tcp_ka_abort_thres / *i1) >
779                                 tcp->tcp_rto_max)
780                                     return (EINVAL);

782                             tcp->tcp_ka_rinterval =
783                                 tcp->tcp_ka_abort_thres / *i1;
784                     } else {
785                             if ((*i1 * tcp->tcp_ka_rinterval) <
786                                 tcps->tcps_keepalive_abort_interval_low ||
787                                 (*i1 * tcp->tcp_ka_rinterval) >
```

```
788                                     tcps->tcps_keepalive_abort_interval_high)
789                                         return (EINVAL);
790                                 tcp->tcp_ka_abort_thres =
791                                     (*i1 * tcp->tcp_ka_rinterval);
792                         }
793                         tcp->tcp_ka_cnt = *i1;
794                         break;
795                 case TCP_KEEPINTVL:
796                         /*
797                          * TCP_KEEPINTVL is specified in seconds, but
798                          * tcp_ka_rinterval is in milliseconds.
799                          */
801                         if (checkonly)
802                                 break;
804                         if ((*i1 * 1000) < tcp->tcp_rto_min ||
805                             (*i1 * 1000) > tcp->tcp_rto_max)
806                                 return (EINVAL);
808                         if (tcp->tcp_ka_cnt == 0) {
809                                 tcp->tcp_ka_cnt =
810                                     tcp->tcp_ka_abort_thres / (*i1 * 1000);
811                         } else {
812                                 if ((*i1 * tcp->tcp_ka_cnt * 1000) <
813                                     tcps->tcps_keepalive_abort_interval_low ||
814                                     (*i1 * tcp->tcp_ka_cnt * 1000) >
815                                     tcps->tcps_keepalive_abort_interval_high)
816                                         return (EINVAL);
817                                 tcp->tcp_ka_abort_thres =
818                                     (*i1 * tcp->tcp_ka_cnt * 1000);
819                         }
820                         tcp->tcp_ka_rinterval = *i1 * 1000;
821                         break;
822                 case TCP_KEEPALIVE_ABORT_THRESHOLD:
823                         if (!checkonly) {
824                                 if (*i1 <
825                                     tcps->tcps_keepalive_abort_interval_low ||
826                                     *i1 >
827                                     tcps->tcps_keepalive_abort_interval_high) {
828                                         *outlenp = 0;
829                                         return (EINVAL);
830                                 }
831                                 tcp->tcp_ka_abort_thres = *i1;
832                                 tcp->tcp_ka_cnt = 0;
833                                 tcp->tcp_ka_rinterval = 0;
834                         }
835                         break;
836                 case TCP_CORK:
837                         if (!checkonly) {
838                                 /*
839                                  * if tcp->tcp_cork was set and is now
840                                  * being unset, we have to make sure that
841                                  * the remaining data gets sent out. Also
842                                  * unset tcp->tcp_cork so that tcp_wput_data()
843                                  * can send data even if it is less than mss
844                                  */
845                                 if (tcp->tcp_cork && onoff == 0 &&
846                                     tcp->tcp_unsent > 0) {
847                                         tcp->tcp_cork = B_FALSE;
848                                         tcp_wput_data(tcp, NULL, B_FALSE);
849                                 }
850                                 tcp->tcp_cork = onoff;
851                         }
852                         break;
853                 case TCP_RTO_INITIAL: {
```



```
854                         clock_t rto;
856                         if (checkonly || val == 0)
857                                 break;
859                         /*
860                          * Sanity checks
861                          *
862                          * The initial RTO should be bounded by the minimum
863                          * and maximum RTO.  And it should also be smaller
864                          * than the connect attempt abort timeout.  Otherwise,
865                          * the connection won't be aborted in a period
866                          * reasonably close to that timeout.
867                          */
868                         if (val < tcp->tcp_rto_min || val > tcp->tcp_rto_max ||
869                             val > tcp->tcp_second_ctimer_threshold ||
870                             val < tcps->tcps_rexmit_interval_initial_low ||
871                             val > tcps->tcps_rexmit_interval_initial_high) {
872                                 *outlenp = 0;
873                                 return (EINVAL);
874                         }
875                         tcp->tcp_rto_initial = val;
877                         /*
878                          * If TCP has not sent anything, need to re-calculate
879                          * tcp_rto.  Otherwise, this option change does not
880                          * really affect anything.
881                          */
882                         if (tcp->tcp_state >= TCPS_SYN_SENT)
883                                 break;
885                         tcp->tcp_rtt_sa = tcp->tcp_rto_initial << 2;
886                         tcp->tcp_rtt_sd = tcp->tcp_rto_initial >> 1;
887                         rto = (tcp->tcp_rtt_sa >> 3) + tcp->tcp_rtt_sd +
888                             tcps->tcps_rexmit_interval_extra +
889                             (tcp->tcp_rtt_sa >> 5) +
890                             tcps->tcps_conn_grace_period;
891                         TCP_SET_RTO(tcp, rto);
892                         break;
893                 }
894                 case TCP_RTO_MIN:
895                         if (checkonly || val == 0)
896                                 break;
898                         if (val < tcps->tcps_rexmit_interval_min_low ||
899                             val > tcps->tcps_rexmit_interval_min_high ||
900                             val > tcp->tcp_rto_max) {
901                                 *outlenp = 0;
902                                 return (EINVAL);
903                         }
904                         tcp->tcp_rto_min = val;
905                         if (tcp->tcp_rto < val)
906                                 tcp->tcp_rto = val;
907                         break;
908                 case TCP_RTO_MAX:
909                         if (checkonly || val == 0)
910                                 break;
912                         /*
913                          * Sanity checks
914                          *
915                          * The maximum RTO should not be larger than the
916                          * connection abort timeout.  Otherwise, the
917                          * connection won't be aborted in a period reasonably
918                          * close to that timeout.
919                          */
```

```
920                     if (val < tcps->tcps_rexmit_interval_max_low ||
921                         val > tcps->tcps_rexmit_interval_max_high ||
922                         val < tcp->tcp_rto_min ||
923                         val > tcp->tcp_second_timer_threshold) {
924                             *outlenp = 0;
925                             return (EINVAL);
926                     }
927                     tcp->tcp_rto_max = val;
928                     if (tcp->tcp_rto > val)
929                             tcp->tcp_rto = val;
930                     break;
931             case TCP_LINGER2:
932                     if (checkonly || *i1 == 0)
933                             break;

935                     /*
936                      * Note that the option value's unit is second.  And
937                      * the value should be bigger than the private
938                      * parameter tcp_fin_wait_2_flush_interval's lower
939                      * bound and smaller than the current value of that
940                      * parameter.  It should be smaller than the current
941                      * value to avoid an app setting TCP_LINGER2 to a big
942                      * value, causing resource to be held up too long in
943                      * FIN-WAIT-2 state.
944                      */
945                     if (*i1 < 0 ||
946                         tcps->tcps_fin_wait_2_flush_interval_low/SECONDS >
947                         *i1 ||
948                         tcps->tcps_fin_wait_2_flush_interval/SECONDS <
949                         *i1) {
950                             *outlenp = 0;
951                             return (EINVAL);
952                     }
953                     tcp->tcp_fin_wait_2_flush_interval = *i1 * SECONDS;
954                     break;
955             default:
956                     break;
957             }
958             break;
959     case IPPROTO_IP:
960             if (connp->conn_family != AF_INET) {
961                     *outlenp = 0;
962                     return (EINVAL);
963             }
964             switch (name) {
965             case IP_SEC_OPT:
966                     /*
967                      * We should not allow policy setting after
968                      * we start listening for connections.
969                      */
970                     if (tcp->tcp_state == TCPS_LISTEN) {
971                             return (EINVAL);
972                     }
973                     break;
974             }
975             break;
976     case IPPROTO_IPV6:
977             /*
978              * IPPROTO_IPV6 options are only supported for sockets
979              * that are using IPv6 on the wire.
980              */
981             if (connp->conn_ipversion != IPV6_VERSION) {
982                     *outlenp = 0;
983                     return (EINVAL);
984             }
```



```
986             switch (name) {
987             case IPV6_RECVPKTINFO:
988                     if (!checkonly) {
989                             /* Force it to be sent up with the next msg */
990                             tcp->tcp_recvifindex = 0;
991                     }
992                     break;
993             case IPV6_RECVTCLASS:
994                     if (!checkonly) {
995                             /* Force it to be sent up with the next msg */
996                             tcp->tcp_recvtclass = 0xffffffffU;
997                     }
998                     break;
999             case IPV6_RECVHOPLIMIT:
1000                    if (!checkonly) {
1001                            /* Force it to be sent up with the next msg */
1002                            tcp->tcp_recvhops = 0xffffffffU;
1003                    }
1004                    break;
1005            case IPV6_PKTINFO:
1006                    /* This is an extra check for TCP */
1007                    if (inlen == sizeof (struct in6_pktinfo)) {
1008                            struct in6_pktinfo *pkti;

1010                            pkti = (struct in6_pktinfo *)invalp;
1011                            /*
1012                             * RFC 3542 states that ipi6_addr must be
1013                             * the unspecified address when setting the
1014                             * IPV6_PKTINFO sticky socket option on a
1015                             * TCP socket.
1016                             */
1017                            if (!IN6_IS_ADDR_UNSPECIFIED(&pkti->ipi6_addr))
1018                                    return (EINVAL);
1019                    }
1020                    break;
1021            case IPV6_SEC_OPT:
1022                    /*
1023                     * We should not allow policy setting after
1024                     * we start listening for connections.
1025                     */
1026                    if (tcp->tcp_state == TCPS_LISTEN) {
1027                            return (EINVAL);
1028                    }
1029                    break;
1030            }
1031            break;
1032    }
1033    reterr = conn_opt_set(&coas, level, name, inlen, invalp,
1034        checkonly, cr);
1035    if (reterr != 0) {
1036            *outlenp = 0;
1037            return (reterr);
1038    }

1040    /*
1041     * Common case of OK return with outval same as inval
1042     */
1043    if (invalp != outvalp) {
1044            /* don't trust bcopy for identical src/dst */
1045            (void) bcopy(invalp, outvalp, inlen);
1046    }
1047    *outlenp = inlen;

1049    if (coas.coa_changed & COA_HEADER_CHANGED) {
1050            /* If we are connected we rebuilt the headers */
1051            if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) &&
```

```
1052                     !IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
1053                         reterr = tcp_build_hdrs(tcp);
1054                         if (reterr != 0)
1055                                 return (reterr);
1056                 }
1057         }
1058         if (coas.coa_changed & COA_ROUTE_CHANGED) {
1059                 in6_addr_t nexthop;

1061                 /*
1062                  * If we are connected we re-cache the information.
1063                  * We ignore errors to preserve BSD behavior.
1064                  * Note that we don't redo IPsec policy lookup here
1065                  * since the final destination (or source) didn't change.
1066                  */
1067                 ip_attr_nexthop(&connp->conn_xmit_ipp, connp->conn_ixa,
1068                     &connp->conn_faddr_v6, &nexthop);

1070                 if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) &&
1071                     !IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
1072                         (void) ip_attr_connect(connp, connp->conn_ixa,
1073                             &connp->conn_laddr_v6, &connp->conn_faddr_v6,
1074                             &nexthop, connp->conn_fport, NULL, NULL,
1075                             IPDF_VERIFY_DST);
1076                 }
1077         }
1078         if ((coas.coa_changed & COA_SNDBUF_CHANGED) && !IPCL_IS_NONSTR(connp)) {
1079                 connp->conn_wq->q_hiwat = connp->conn_sndbuf;
1080         }
1081         if (coas.coa_changed & COA_WROFF_CHANGED) {
1082                 connp->conn_wroff = connp->conn_ht_iphc_allocated +
1083                     tcps->tcps_wroff_xtra;
1084                 (void) proto_set_tx_wroff(connp->conn_rq, connp,
1085                     connp->conn_wroff);
1086         }
1087         if (coas.coa_changed & COA_OOBINLINE_CHANGED) {
1088                 if (IPCL_IS_NONSTR(connp))
1089                         proto_set_rx_oob_opt(connp, onoff);
1090         }
1091         return (0);
1092 }
```

```
**********************************************************
   31926 Mon Jul  9 14:38:20 2012
new/usr/src/uts/common/inet/tcp/tcp_socket.c
tcp: maybe related to 721fffe3
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 */

/* This file contains all TCP kernel socket related functions. */

#include <sys/types.h>
#include <sys/strlog.h>
#include <sys/policy.h>
#include <sys/sockio.h>
#include <sys/strsubr.h>
#include <sys/strsun.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/timod.h>
#include <sys/tpicommon.h>
#include <sys/socketvar.h>

#include <inet/common.h>
#include <inet/proto_set.h>
#include <inet/ip.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>

static void     tcp_activate(sock_lower_handle_t, sock_upper_handle_t,
                    sock_upcalls_t *, int, cred_t *);
static int      tcp_accept(sock_lower_handle_t, sock_lower_handle_t,
                    sock_upper_handle_t, cred_t *);
static int      tcp_bind(sock_lower_handle_t, struct sockaddr *,
                    socklen_t, cred_t *);
static int      tcp_listen(sock_lower_handle_t, int, cred_t *);
static int      tcp_connect(sock_lower_handle_t, const struct sockaddr *,
                    socklen_t, sock_connid_t *, cred_t *);
static int      tcp_getpeername(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
static int      tcp_getsockname(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
#endif /* ! codereview */
```



```
static int      tcp_getsockopt(sock_lower_handle_t, int, int, void *,
                    socklen_t *, cred_t *);
static int      tcp_setsockopt(sock_lower_handle_t, int, int, const void *,
                    socklen_t, cred_t *);
static int      tcp_sendmsg(sock_lower_handle_t, mblk_t *, struct nmsghdr *,
                    cred_t *);
                    cred_t *cr);
static int      tcp_shutdown(sock_lower_handle_t, int, cred_t *);
static void     tcp_clr_flowctrl(sock_lower_handle_t);
static int      tcp_ioctl(sock_lower_handle_t, int, intptr_t, int, int32_t *,
                    cred_t *);
static int      tcp_close(sock_lower_handle_t, int, cred_t *);

sock_downcalls_t sock_tcp_downcalls = {
        tcp_activate,
        tcp_accept,
        tcp_bind,
        tcp_listen,
        tcp_connect,
        tcp_getpeername,
        tcp_getsockname,
        tcp_getsockopt,
        tcp_setsockopt,
        tcp_sendmsg,
        NULL,
        NULL,
        NULL,
        tcp_shutdown,
        tcp_clr_flowctrl,
        tcp_ioctl,
        tcp_close,
};
```

*unchanged_portion_omitted_*

```
/* ARGSUSED */
sock_lower_handle_t
tcp_create(int family, int type, int proto, sock_downcalls_t **sock_downcalls,
    uint_t *smodep, int *errorp, int flags, cred_t *credp)
{
        conn_t          *connp;
        boolean_t       isv6 = family == AF_INET6;

#endif /* ! codereview */
        if (type != SOCK_STREAM || (family != AF_INET && family != AF_INET6) ||
            (proto != 0 && proto != IPPROTO_TCP)) {
                *errorp = EPROTONOSUPPORT;
                return (NULL);
        }

        connp = tcp_create_common(credp, isv6, B_TRUE, errorp);
        if (connp == NULL) {
                return (NULL);
        }

        /*
         * Put the ref for TCP. Ref for IP was already put
         * by ipcl_conn_create. Also make the conn_t globally
         * visible to walkers.
         * by ipcl_conn_create. Also Make the conn_t globally
         * visible to walkers
         */
        mutex_enter(&connp->conn_lock);
        CONN_INC_REF_LOCKED(connp);
        ASSERT(connp->conn_ref == 2);
        connp->conn_state_flags &= ~CONN_INCIPIENT;
```

```
 783         connp->conn_flags |= IPCL_NONSTR;
 784         mutex_exit(&connp->conn_lock);

 786         ASSERT(errorp != NULL);
 787         *errorp = 0;
 788         *sock_downcalls = &sock_tcp_downcalls;
 789         *smodep = SM_CONNREQUIRED | SM_EXDATA | SM_ACCEPTSUPP |
 790             SM_SENDFILESUPP;

 792         return ((sock_lower_handle_t)connp);
 793 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28387 Mon Jul  9 14:38:21 2012
new/usr/src/uts/common/inet/tcp_impl.h
tcp: maybe related to 721fffe3
**********************************************************
______unchanged_portion_omitted_

 339 /* Increment and decrement the number of connections in tcp_stack_t. */
 340 #define TCPS_CONN_INC(tcps)                                             \
 341         atomic_inc_64(                                                  \
 342             (uint64_t *)&(tcps)->tcps_sc[CPU->cpu_seqid]->tcp_sc_conn_cnt)

 344 #define TCPS_CONN_DEC(tcps)                                             \
 345         atomic_dec_64(                                                  \
 346             (uint64_t *)&(tcps)->tcps_sc[CPU->cpu_seqid]->tcp_sc_conn_cnt)

 348 /*
 349  * When the system is under memory pressure, stack variable tcps_reclaim is
 350  * true, we shorten the connection timeout abort interval to tcp_early_abort
 351  * seconds.  Defined in tcp.c.
 352  */
 353 extern uint32_t tcp_early_abort;

 355 /*
 356  * To reach to an eager in Q0 which can be dropped due to an incoming
 357  * new SYN request when Q0 is full, a new doubly linked list is
 358  * introduced. This list allows to select an eager from Q0 in O(1) time.
 359  * This is needed to avoid spending too much time walking through the
 360  * long list of eagers in Q0 when tcp_drop_q0() is called. Each member of
 361  * this new list has to be a member of Q0.
 362  * This list is headed by listener's tcp_t. When the list is empty,
 363  * both the pointers - tcp_eager_next_drop_q0 and tcp_eager_prev_drop_q0,
 364  * of listener's tcp_t point to listener's tcp_t itself.
 365  *
 366  * Given an eager in Q0 and a listener, MAKE_DROPPABLE() puts the eager
 367  * in the list. MAKE_UNDROPPABLE() takes the eager out of the list.
 368  * These macros do not affect the eager's membership to Q0.
 369  */
 370 #define MAKE_DROPPABLE(listener, eager)                                 \
 371         if ((eager)->tcp_eager_next_drop_q0 == NULL) {                  \
 372                 (listener)->tcp_eager_next_drop_q0->tcp_eager_prev_drop_q0\
 373                     = (eager);                                          \
 374                 (eager)->tcp_eager_prev_drop_q0 = (listener);           \
 375                 (eager)->tcp_eager_next_drop_q0 =                       \
 376                     (listener)->tcp_eager_next_drop_q0;                 \
 377                 (listener)->tcp_eager_next_drop_q0 = (eager);           \
 378         }

 380 #define MAKE_UNDROPPABLE(eager)                                         \
 381         if ((eager)->tcp_eager_next_drop_q0 != NULL) {                  \
 382                 (eager)->tcp_eager_next_drop_q0->tcp_eager_prev_drop_q0 \
 383                     = (eager)->tcp_eager_prev_drop_q0;                  \
 384                 (eager)->tcp_eager_prev_drop_q0->tcp_eager_next_drop_q0 \
 385                     = (eager)->tcp_eager_next_drop_q0;                  \
 386                 (eager)->tcp_eager_prev_drop_q0 = NULL;                 \
 387                 (eager)->tcp_eager_next_drop_q0 = NULL;                 \
 388         }

 390 /*
 391  * The format argument to pass to tcp_display().
 392  * DISP_PORT_ONLY means that the returned string has only port info.
 393  * DISP_ADDR_AND_PORT means that the returned string also contains the
 394  * remote and local IP address.
 395  */
 396 #define DISP_PORT_ONLY          1
 397 #define DISP_ADDR_AND_PORT      2
```



```
 399 #define IP_ADDR_CACHE_SIZE      2048
 400 #define IP_ADDR_CACHE_HASH(faddr)                                       \
 401         (ntohl(faddr) & (IP_ADDR_CACHE_SIZE -1))

 403 /* TCP cwnd burst factor. */
 404 #define TCP_CWND_INFINITE       65535
 405 #define TCP_CWND_SS             3
 406 #define TCP_CWND_NORMAL         5

 408 /*
 409  * TCP reassembly macros.  We hide starting and ending sequence numbers in
 410  * b_next and b_prev of messages on the reassembly queue.  The messages are
 411  * chained using b_cont.  These macros are used in tcp_reass() so we don't
 412  * have to see the ugly casts and assignments.
 413  */
 414 #define TCP_REASS_SEQ(mp)               ((uint32_t)(uintptr_t)((mp)->b_next))
 415 #define TCP_REASS_SET_SEQ(mp, u)        ((mp)->b_next = \
 416                                         (mblk_t *)(uintptr_t)(u))
 417 #define TCP_REASS_END(mp)               ((uint32_t)(uintptr_t)((mp)->b_prev))
 418 #define TCP_REASS_SET_END(mp, u)        ((mp)->b_prev = \
 419                                         (mblk_t *)(uintptr_t)(u))

 421 #define tcps_time_wait_interval         tcps_propinfo_tbl[0].prop_cur_uval
 422 #define tcps_conn_req_max_q             tcps_propinfo_tbl[1].prop_cur_uval
 423 #define tcps_conn_req_max_q0            tcps_propinfo_tbl[2].prop_cur_uval
 424 #define tcps_conn_req_min               tcps_propinfo_tbl[3].prop_cur_uval
 425 #define tcps_conn_grace_period          tcps_propinfo_tbl[4].prop_cur_uval
 426 #define tcps_cwnd_max_                  tcps_propinfo_tbl[5].prop_cur_uval
 427 #define tcps_dbg                        tcps_propinfo_tbl[6].prop_cur_uval
 428 #define tcps_smallest_nonpriv_port      tcps_propinfo_tbl[7].prop_cur_uval
 429 #define tcps_ip_abort_cinterval         tcps_propinfo_tbl[8].prop_cur_uval
 430 #define tcps_ip_abort_linterval         tcps_propinfo_tbl[9].prop_cur_uval
 431 #define tcps_ip_abort_interval          tcps_propinfo_tbl[10].prop_cur_uval
 432 #define tcps_ip_notify_cinterval        tcps_propinfo_tbl[11].prop_cur_uval
 433 #define tcps_ip_notify_interval         tcps_propinfo_tbl[12].prop_cur_uval
 434 #define tcps_ipv4_ttl                   tcps_propinfo_tbl[13].prop_cur_uval
 435 #define tcps_keepalive_interval_high    tcps_propinfo_tbl[14].prop_max_uval
 436 #define tcps_keepalive_interval         tcps_propinfo_tbl[14].prop_cur_uval
 437 #define tcps_keepalive_interval_low     tcps_propinfo_tbl[14].prop_min_uval
 438 #define tcps_maxpsz_multiplier          tcps_propinfo_tbl[15].prop_cur_uval
 439 #define tcps_mss_def_ipv4               tcps_propinfo_tbl[16].prop_cur_uval
 440 #define tcps_mss_max_ipv4               tcps_propinfo_tbl[17].prop_cur_uval
 441 #define tcps_mss_min                    tcps_propinfo_tbl[18].prop_cur_uval
 442 #define tcps_naglim_def                 tcps_propinfo_tbl[19].prop_cur_uval
 443 #define tcps_rexmit_interval_initial_high       \
 444                                         tcps_propinfo_tbl[20].prop_max_uval
 445 #define tcps_rexmit_interval_initial    tcps_propinfo_tbl[20].prop_cur_uval
 446 #define tcps_rexmit_interval_initial_low        \
 447                                         tcps_propinfo_tbl[20].prop_min_uval
 448 #define tcps_rexmit_interval_max_high   tcps_propinfo_tbl[21].prop_max_uval
 449 #define tcps_rexmit_interval_max        tcps_propinfo_tbl[21].prop_cur_uval
 450 #define tcps_rexmit_interval_max_low    tcps_propinfo_tbl[21].prop_min_uval
 451 #define tcps_rexmit_interval_min_high   tcps_propinfo_tbl[22].prop_max_uval
 452 #define tcps_rexmit_interval_min        tcps_propinfo_tbl[22].prop_cur_uval
 453 #define tcps_rexmit_interval_min_low    tcps_propinfo_tbl[22].prop_min_uval
 454 #define tcps_deferred_ack_interval      tcps_propinfo_tbl[23].prop_cur_uval
 455 #define tcps_snd_lowat_fraction         tcps_propinfo_tbl[24].prop_cur_uval
 456 #define tcps_dupack_fast_retransmit     tcps_propinfo_tbl[25].prop_cur_uval
 457 #define tcps_ignore_path_mtu            tcps_propinfo_tbl[26].prop_cur_bval
 458 #define tcps_smallest_anon_port         tcps_propinfo_tbl[27].prop_cur_uval
 459 #define tcps_largest_anon_port          tcps_propinfo_tbl[28].prop_cur_uval
 460 #define tcps_xmit_hiwat                 tcps_propinfo_tbl[29].prop_cur_uval
 461 #define tcps_xmit_lowat                 tcps_propinfo_tbl[30].prop_cur_uval
 462 #define tcps_recv_hiwat                 tcps_propinfo_tbl[31].prop_cur_uval
 463 #define tcps_recv_hiwat_minmss          tcps_propinfo_tbl[32].prop_cur_uval
```

```
#define tcps_fin_wait_2_flush_interval_high     \
                                        tcps_propinfo_tbl[33].prop_max_uval
#define tcps_fin_wait_2_flush_interval  tcps_propinfo_tbl[33].prop_cur_uval
#define tcps_fin_wait_2_flush_interval_low      \
                                        tcps_propinfo_tbl[33].prop_min_uval
#define tcps_max_buf                    tcps_propinfo_tbl[34].prop_cur_uval
#define tcps_strong_iss                 tcps_propinfo_tbl[35].prop_cur_uval
#define tcps_rtt_updates                tcps_propinfo_tbl[36].prop_cur_uval
#define tcps_wscale_always              tcps_propinfo_tbl[37].prop_cur_bval
#define tcps_tstamp_always              tcps_propinfo_tbl[38].prop_cur_bval
#define tcps_tstamp_if_wscale           tcps_propinfo_tbl[39].prop_cur_bval
#define tcps_rexmit_interval_extra      tcps_propinfo_tbl[40].prop_cur_uval
#define tcps_deferred_acks_max          tcps_propinfo_tbl[41].prop_cur_uval
#define tcps_slow_start_after_idle      tcps_propinfo_tbl[42].prop_cur_uval
#define tcps_slow_start_initial         tcps_propinfo_tbl[43].prop_cur_uval
#define tcps_sack_permitted             tcps_propinfo_tbl[44].prop_cur_uval
#define tcps_ipv6_hoplimit              tcps_propinfo_tbl[45].prop_cur_uval
#define tcps_mss_def_ipv6               tcps_propinfo_tbl[46].prop_cur_uval
#define tcps_mss_max_ipv6               tcps_propinfo_tbl[47].prop_cur_uval
#define tcps_rev_src_routes             tcps_propinfo_tbl[48].prop_cur_bval
#define tcps_local_dack_interval        tcps_propinfo_tbl[49].prop_cur_uval
#define tcps_local_dacks_max            tcps_propinfo_tbl[50].prop_cur_uval
#define tcps_ecn_permitted              tcps_propinfo_tbl[51].prop_cur_uval
#define tcps_rst_sent_rate_enabled      tcps_propinfo_tbl[52].prop_cur_bval
#define tcps_rst_sent_rate              tcps_propinfo_tbl[53].prop_cur_uval
#define tcps_push_timer_interval        tcps_propinfo_tbl[54].prop_cur_uval
#define tcps_use_smss_as_mss_opt        tcps_propinfo_tbl[55].prop_cur_bval
#define tcps_keepalive_abort_interval_high \
                                        tcps_propinfo_tbl[56].prop_max_uval
#define tcps_keepalive_abort_interval \
                                        tcps_propinfo_tbl[56].prop_cur_uval
#define tcps_keepalive_abort_interval_low \
                                        tcps_propinfo_tbl[56].prop_min_uval
#define tcps_wroff_xtra                 tcps_propinfo_tbl[57].prop_cur_uval
#define tcps_dev_flow_ctl               tcps_propinfo_tbl[58].prop_cur_bval
#define tcps_reass_timeout              tcps_propinfo_tbl[59].prop_cur_uval
#define tcps_iss_incr                   tcps_propinfo_tbl[65].prop_cur_uval

extern struct qinit tcp_rinitv4, tcp_rinitv6;
extern boolean_t do_tcp_fusion;

/*
 * Object to represent database of options to search passed to
 * {sock,tpi}optcom_req() interface routine to take care of option
 * management and associated methods.
 */
extern optdb_obj_t      tcp_opt_obj;
extern uint_t           tcp_max_optsize;

extern int tcp_squeue_flag;

extern uint_t tcp_free_list_max_cnt;

/*
 * Functions in tcp.c.
 */
extern void     tcp_acceptor_hash_insert(t_uscalar_t, tcp_t *);
extern tcp_t    *tcp_acceptor_hash_lookup(t_uscalar_t, tcp_stack_t *);
extern void     tcp_acceptor_hash_remove(tcp_t *);
extern mblk_t   *tcp_ack_mp(tcp_t *);
extern int      tcp_build_hdrs(tcp_t *);
extern void     tcp_cleanup(tcp_t *);
extern int      tcp_clean_death(tcp_t *, int);
extern void     tcp_clean_death_wrapper(void *, mblk_t *, void *,
                    ip_recv_attr_t *);
extern void     tcp_close_common(conn_t *, int);
```



```
extern void     tcp_close_detached(tcp_t *);
extern void     tcp_close_mpp(mblk_t **);
extern void     tcp_closei_local(tcp_t *);
extern sock_lower_handle_t tcp_create(int, int, int, sock_downcalls_t **,
                    uint_t *, int *, int, cred_t *);
extern conn_t   *tcp_create_common(cred_t *, boolean_t, boolean_t, int *);
extern void     tcp_disconnect(tcp_t *, mblk_t *);
extern char     *tcp_display(tcp_t *, char *, char);
extern int      tcp_do_bind(conn_t *, struct sockaddr *, socklen_t, cred_t *,
                    boolean_t);
extern int      tcp_do_connect(conn_t *, const struct sockaddr *, socklen_t,
                    cred_t *, pid_t);
extern int      tcp_do_listen(conn_t *, struct sockaddr *, socklen_t, int,
                    cred_t *, boolean_t);
extern int      tcp_do_unbind(conn_t *);
extern boolean_t        tcp_eager_blowoff(tcp_t *, t_scalar_t);
extern void     tcp_eager_cleanup(tcp_t *, boolean_t);
extern void     tcp_eager_kill(void *, mblk_t *, void *, ip_recv_attr_t *);
extern void     tcp_eager_unlink(tcp_t *);
extern int      tcp_getpeername(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
extern int      tcp_getsockname(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
extern void     tcp_init_values(tcp_t *, tcp_t *);
extern void     tcp_ipsec_cleanup(tcp_t *);
extern int      tcp_maxpsz_set(tcp_t *, boolean_t);
extern void     tcp_mss_set(tcp_t *, uint32_t);
extern void     tcp_reinput(conn_t *, mblk_t *, ip_recv_attr_t *, ip_stack_t *);
extern void     tcp_rsrv(queue_t *);
extern uint_t   tcp_rwnd_reopen(tcp_t *);
extern int      tcp_rwnd_set(tcp_t *, uint32_t);
extern int      tcp_set_destination(tcp_t *);
extern void     tcp_set_ws_value(tcp_t *);
extern void     tcp_stop_lingering(tcp_t *);
extern void     tcp_update_pmtu(tcp_t *, boolean_t);
extern mblk_t   *tcp_zcopy_backoff(tcp_t *, mblk_t *, boolean_t);
extern boolean_t        tcp_zcopy_check(tcp_t *);
extern void     tcp_zcopy_notify(tcp_t *);
extern void     tcp_get_proto_props(tcp_t *, struct sock_proto_props *);

/*
 * Bind related functions in tcp_bind.c
 */
extern int      tcp_bind_check(conn_t *, struct sockaddr *, socklen_t,
                    cred_t *, boolean_t);
extern void     tcp_bind_hash_insert(tf_t *, tcp_t *, int);
extern void     tcp_bind_hash_remove(tcp_t *);
extern in_port_t        tcp_bindi(tcp_t *, in_port_t, const in6_addr_t *,
                            int, boolean_t, boolean_t, boolean_t);
extern in_port_t        tcp_update_next_port(in_port_t, const tcp_t *,
                            boolean_t);

/*
 * Fusion related functions in tcp_fusion.c.
 */
extern void     tcp_fuse(tcp_t *, uchar_t *, tcpha_t *);
extern void     tcp_unfuse(tcp_t *);
extern boolean_t tcp_fuse_output(tcp_t *, mblk_t *, uint32_t);
extern void     tcp_fuse_output_urg(tcp_t *, mblk_t *);
extern boolean_t tcp_fuse_rcv_drain(queue_t *, tcp_t *, mblk_t **);
extern size_t   tcp_fuse_set_rcv_hiwat(tcp_t *, size_t);
extern int      tcp_fuse_maxpsz(tcp_t *);
extern void     tcp_fuse_backenable(tcp_t *);
extern void     tcp_iss_key_init(uint8_t *, int, tcp_stack_t *);

/*
```

```
 592  * Output related functions in tcp_output.c.
 593  */
 594 extern void     tcp_close_output(void *, mblk_t *, void *, ip_recv_attr_t *);
 595 extern void     tcp_output(void *, mblk_t *, void *, ip_recv_attr_t *);
 596 extern void     tcp_output_urgent(void *, mblk_t *, void *, ip_recv_attr_t *);
 597 extern void     tcp_rexmit_after_error(tcp_t *);
 598 extern void     tcp_sack_rexmit(tcp_t *, uint_t *);
 599 extern void     tcp_send_data(tcp_t *, mblk_t *);
 600 extern void     tcp_send_synack(void *, mblk_t *, void *, ip_recv_attr_t *);
 601 extern void     tcp_shutdown_output(void *, mblk_t *, void *, ip_recv_attr_t *);
 602 extern void     tcp_ss_rexmit(tcp_t *);
 603 extern void     tcp_update_xmit_tail(tcp_t *, uint32_t);
 604 extern void     tcp_wput(queue_t *, mblk_t *);
 605 extern void     tcp_wput_data(tcp_t *, mblk_t *, boolean_t);
 606 extern void     tcp_wput_sock(queue_t *, mblk_t *);
 607 extern void     tcp_wput_fallback(queue_t *, mblk_t *);
 608 extern void     tcp_xmit_ctl(char *, tcp_t *, uint32_t, uint32_t, int);
 609 extern void     tcp_xmit_listeners_reset(mblk_t *, ip_recv_attr_t *,
 610                     ip_stack_t *i, conn_t *);
 611 extern mblk_t   *tcp_xmit_mp(tcp_t *, mblk_t *, int32_t, int32_t *,
 612                     mblk_t **, uint32_t, boolean_t, uint32_t *, boolean_t);

 614 /*
 615  * Input related functions in tcp_input.c.
 616  */
 617 extern void     tcp_icmp_input(void *, mblk_t *, void *, ip_recv_attr_t *);
 618 extern void     tcp_input_data(void *, mblk_t *, void *, ip_recv_attr_t *);
 619 extern void     tcp_input_listener_unbound(void *, mblk_t *, void *,
 620                     ip_recv_attr_t *);
 621 extern boolean_t        tcp_paws_check(tcp_t *, tcpha_t *, tcp_opt_t *);
 622 extern uint_t   tcp_rcv_drain(tcp_t *);
 623 extern void     tcp_rcv_enqueue(tcp_t *, mblk_t *, uint_t, cred_t *);
 624 extern boolean_t        tcp_verifyicmp(conn_t *, void *, icmph_t *, icmp6_t *,
 625                             ip_recv_attr_t *);

 627 /*
 628  * Kernel socket related functions in tcp_socket.c.
 629  */
 630 extern int      tcp_fallback(sock_lower_handle_t, queue_t *, boolean_t,
 631                     so_proto_quiesced_cb_t, sock_quiesce_arg_t *);
 632 extern boolean_t tcp_newconn_notify(tcp_t *, ip_recv_attr_t *);

 634 /*
 635  * Timer related functions in tcp_timers.c.
 636  */
 637 extern void     tcp_ack_timer(void *);
 638 extern void     tcp_close_linger_timeout(void *);
 639 extern void     tcp_keepalive_timer(void *);
 640 extern void     tcp_push_timer(void *);
 641 extern void     tcp_reass_timer(void *);
 642 extern mblk_t   *tcp_timermp_alloc(int);
 643 extern void     tcp_timermp_free(tcp_t *);
 644 extern timeout_id_t tcp_timeout(conn_t *, void (*)(void *), hrtime_t);
 645 extern clock_t  tcp_timeout_cancel(conn_t *, timeout_id_t);
 646 extern void     tcp_timer(void *arg);
 647 extern void     tcp_timers_stop(tcp_t *);

 649 /*
 650  * TCP TPI related functions in tcp_tpi.c.
 651  */
 652 extern void     tcp_addr_req(tcp_t *, mblk_t *);
 653 extern void     tcp_capability_req(tcp_t *, mblk_t *);
 654 extern boolean_t        tcp_conn_con(tcp_t *, uchar_t *, mblk_t *,
 655                             mblk_t **, ip_recv_attr_t *);
 656 extern void     tcp_err_ack(tcp_t *, mblk_t *, int, int);
 657 extern void     tcp_err_ack_prim(tcp_t *, mblk_t *, int, int, int);
```



```
 658 extern void     tcp_info_req(tcp_t *, mblk_t *);
 659 extern void     tcp_send_conn_ind(void *, mblk_t *, void *);
 660 extern void     tcp_send_pending(void *, mblk_t *, void *, ip_recv_attr_t *);
 661 extern void     tcp_tpi_accept(queue_t *, mblk_t *);
 662 extern void     tcp_tpi_bind(tcp_t *, mblk_t *);
 663 extern int      tcp_tpi_close(queue_t *, int);
 664 extern int      tcp_tpi_close_accept(queue_t *);
 665 extern void     tcp_tpi_connect(tcp_t *, mblk_t *);
 666 extern int      tcp_tpi_opt_get(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
 667 extern int      tcp_tpi_opt_set(queue_t *, uint_t, int, int, uint_t, uchar_t *,
 668                     uint_t *, uchar_t *, void *, cred_t *);
 669 extern void     tcp_tpi_unbind(tcp_t *, mblk_t *);
 670 extern void     tcp_tli_accept(tcp_t *, mblk_t *);
 671 extern void     tcp_use_pure_tpi(tcp_t *);
 672 extern void     tcp_do_capability_ack(tcp_t *, struct T_capability_ack *,
 673                     t_uscalar_t);

 675 /*
 676  * TCP option processing related functions in tcp_opt_data.c
 677  */
 682 extern int      tcp_opt_default(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
 678 extern int      tcp_opt_get(conn_t *, int, int, uchar_t *);
 679 extern int      tcp_opt_set(conn_t *, uint_t, int, int, uint_t, uchar_t *,
 680                     uint_t *, uchar_t *, void *, cred_t *);

 682 /*
 683  * TCP time wait processing related functions in tcp_time_wait.c.
 684  */
 685 extern void             tcp_time_wait_append(tcp_t *);
 686 extern void             tcp_time_wait_collector(void *);
 687 extern boolean_t        tcp_time_wait_remove(tcp_t *, tcp_squeue_priv_t *);
 688 extern void             tcp_time_wait_processing(tcp_t *, mblk_t *, uint32_t,
 689                             uint32_t, int, tcpha_t *, ip_recv_attr_t *);

 691 /*
 692  * Misc functions in tcp_misc.c.
 693  */
 694 extern uint32_t tcp_find_listener_conf(tcp_stack_t *, in_port_t);
 695 extern void     tcp_ioctl_abort_conn(queue_t *, mblk_t *);
 696 extern void     tcp_listener_conf_cleanup(tcp_stack_t *);
 697 extern void     tcp_stack_cpu_add(tcp_stack_t *, processorid_t);

 699 #endif  /* _KERNEL */

 701 #ifdef  __cplusplus
 702 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    7689 Mon Jul  9 14:38:21 2012
new/usr/src/uts/common/inet/tcp_stats.h
tcp: spelling
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 */

#ifndef _INET_TCP_STATS_H
#define _INET_TCP_STATS_H

/*
 * TCP private kernel statistics declarations.
 */

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef _KERNEL

/*
 * TCP Statistics.
 *
 * How TCP statistics work.
 *
 * There are two types of statistics invoked by two macros.
 *
 * TCP_STAT(name) does non-atomic increment of a named stat counter. It is
 * supposed to be used in non MT-hot paths of the code.
 *
 * TCP_DBGSTAT(name) does atomic increment of a named stat counter. It is
 * supposed to be used for DEBUG purposes and may be used on a hot path.
 * These counters are only available in a debugged kernel.  They are grouped
 * These counters are only available in a debugged kerel.  They are grouped
 * under the TCP_DEBUG_COUNTER C pre-processor condition.
 *
 * Both TCP_STAT and TCP_DBGSTAT counters are available using kstat
 * (use "kstat tcp" to get them).
 *
 * How to add new counters.
 *
 * 1) Add a field in the tcp_stat structure describing your counter.
 * 2) Add a line in the template in tcp_kstat2_init() with the name
```



```
 *    of the counter.
 * 3) Update tcp_clr_stats() and tcp_cp_stats() with the new counters.
 *    IMPORTANT!! - make sure that all the above functions are in sync !!
 * 4) Use either TCP_STAT or TCP_DBGSTAT with the name.
 *
 * Please avoid using private counters which are not kstat-exported.
 *
 * Implementation note.
 *
 * Both the MIB2 and tcp_stat_t counters are kept per CPU in the array
 * tcps_sc in tcp_stack_t.  Each array element is a pointer to a
 * tcp_stats_cpu_t struct.  Once allocated, the tcp_stats_cpu_t struct is
 * not freed until the tcp_stack_t is going away.  So there is no need to
 * acquire a lock before accessing the stats counters.
 */

#ifndef TCP_DEBUG_COUNTER
#ifdef DEBUG
#define TCP_DEBUG_COUNTER 1
#else
#define TCP_DEBUG_COUNTER 0
#endif
#endif

/* Kstats */
typedef struct tcp_stat {
        kstat_named_t   tcp_time_wait_syn_success;
        kstat_named_t   tcp_clean_death_nondetached;
        kstat_named_t   tcp_eager_blowoff_q;
        kstat_named_t   tcp_eager_blowoff_q0;
        kstat_named_t   tcp_no_listener;
        kstat_named_t   tcp_listendrop;
        kstat_named_t   tcp_listendropq0;
        kstat_named_t   tcp_wsrv_called;
        kstat_named_t   tcp_flwctl_on;
        kstat_named_t   tcp_timer_fire_early;
        kstat_named_t   tcp_timer_fire_miss;
        kstat_named_t   tcp_zcopy_on;
        kstat_named_t   tcp_zcopy_off;
        kstat_named_t   tcp_zcopy_backoff;
        kstat_named_t   tcp_fusion_flowctl;
        kstat_named_t   tcp_fusion_backenabled;
        kstat_named_t   tcp_fusion_urg;
        kstat_named_t   tcp_fusion_putnext;
        kstat_named_t   tcp_fusion_unfusable;
        kstat_named_t   tcp_fusion_aborted;
        kstat_named_t   tcp_fusion_unqualified;
        kstat_named_t   tcp_fusion_rrw_busy;
        kstat_named_t   tcp_fusion_rrw_msgcnt;
        kstat_named_t   tcp_fusion_rrw_plugged;
        kstat_named_t   tcp_in_ack_unsent_drop;
        kstat_named_t   tcp_sock_fallback;
        kstat_named_t   tcp_lso_enabled;
        kstat_named_t   tcp_lso_disabled;
        kstat_named_t   tcp_lso_times;
        kstat_named_t   tcp_lso_pkt_out;
        kstat_named_t   tcp_listen_cnt_drop;
        kstat_named_t   tcp_listen_mem_drop;
        kstat_named_t   tcp_zwin_mem_drop;
        kstat_named_t   tcp_zwin_ack_syn;
        kstat_named_t   tcp_rst_unsent;
        kstat_named_t   tcp_reclaim_cnt;
        kstat_named_t   tcp_reass_timeout;
#ifdef TCP_DEBUG_COUNTER
        kstat_named_t   tcp_time_wait;
        kstat_named_t   tcp_rput_time_wait;
```

```
 127         kstat_named_t   tcp_detach_time_wait;
 128         kstat_named_t   tcp_timeout_calls;
 129         kstat_named_t   tcp_timeout_cached_alloc;
 130         kstat_named_t   tcp_timeout_cancel_reqs;
 131         kstat_named_t   tcp_timeout_canceled;
 132         kstat_named_t   tcp_timermp_freed;
 133         kstat_named_t   tcp_push_timer_cnt;
 134         kstat_named_t   tcp_ack_timer_cnt;
 135 #endif
 136 } tcp_stat_t;
```

unchanged_portion_omitted_

```
**********************************************************
   11297 Mon Jul  9 14:38:21 2012
new/usr/src/uts/common/inet/tunables.c
dccp: starting module template
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 1990 Mentat Inc.
 */

#include <inet/tunables.h>
#include <sys/md5.h>
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <netinet/icmp6.h>
#include <inet/ip_stack.h>
#include <inet/rawip_impl.h>
#include <inet/tcp_stack.h>
#include <inet/tcp_impl.h>
#include <inet/udp_impl.h>
#include <inet/dccp/dccp_stack.h>
#include <inet/dccp/dccp_impl.h>
#endif /* ! codereview */
#include <inet/sctp/sctp_stack.h>
#include <inet/sctp/sctp_impl.h>
#include <inet/tunables.h>

static int
prop_perm2const(mod_prop_info_t *pinfo)
{
        if (pinfo->mpi_setf == NULL)
                return (MOD_PROP_PERM_READ);
        if (pinfo->mpi_getf == NULL)
                return (MOD_PROP_PERM_WRITE);
        return (MOD_PROP_PERM_RW);
}

/*
 * Modifies the value of the property to default value or to the 'pval'
 * specified by the user.
 */
/* ARGSUSED */
int
mod_set_boolean(void *cbarg, cred_t *cr, mod_prop_info_t *pinfo,
    const char *ifname, const void* pval, uint_t flags)
```



```
{
        char            *end;
        unsigned long   new_value;

        if (flags & MOD_PROP_DEFAULT) {
                pinfo->prop_cur_bval = pinfo->prop_def_bval;
                return (0);
        }

        if (ddi_strtoul(pval, &end, 10, &new_value) != 0 || *end != '\0')
                return (EINVAL);
        if (new_value != B_TRUE && new_value != B_FALSE)
                return (EINVAL);
        pinfo->prop_cur_bval = new_value;
        return (0);
}

/*
 * Retrieves property permission, default value, current value or possible
 * values for those properties whose value type is boolean_t.
 */
/* ARGSUSED */
int
mod_get_boolean(void *cbarg, mod_prop_info_t *pinfo, const char *ifname,
    void *pval, uint_t psize, uint_t flags)
{
        boolean_t       get_def = (flags & MOD_PROP_DEFAULT);
        boolean_t       get_perm = (flags & MOD_PROP_PERM);
        boolean_t       get_range = (flags & MOD_PROP_POSSIBLE);
        size_t          nbytes;

        bzero(pval, psize);
        if (get_perm)
                nbytes = snprintf(pval, psize, "%u", prop_perm2const(pinfo));
        else if (get_range)
                nbytes = snprintf(pval, psize, "%u,%u", B_FALSE, B_TRUE);
        else if (get_def)
                nbytes = snprintf(pval, psize, "%u", pinfo->prop_def_bval);
        else
                nbytes = snprintf(pval, psize, "%u", pinfo->prop_cur_bval);
        if (nbytes >= psize)
                return (ENOBUFS);
        return (0);
}

int
mod_uint32_value(const void *pval, mod_prop_info_t *pinfo, uint_t flags,
    ulong_t *new_value)
{
        char            *end;

        if (flags & MOD_PROP_DEFAULT) {
                *new_value = pinfo->prop_def_uval;
                return (0);
        }

        if (ddi_strtoul(pval, &end, 10, (ulong_t *)new_value) != 0 ||
            *end != '\0')
                return (EINVAL);
        if (*new_value < pinfo->prop_min_uval ||
            *new_value > pinfo->prop_max_uval) {
                return (ERANGE);
        }
        return (0);
}
```

```
128 /*
129  * Modifies the value of the property to default value or to the ‘pval’
130  * specified by the user.
131  */
132 /* ARGSUSED */
133 int
134 mod_set_uint32(void *cbarg, cred_t *cr, mod_prop_info_t *pinfo,
135     const char *ifname, const void *pval, uint_t flags)
136 {
137         unsigned long   new_value;
138         int             err;
139
140         if ((err = mod_uint32_value(pval, pinfo, flags, &new_value)) != 0)
141                 return (err);
142         pinfo->prop_cur_uval = (uint32_t)new_value;
143         return (0);
144 }
145
146 /*
147  * Rounds up the value to make it multiple of 8.
148  */
149 /* ARGSUSED */
150 int
151 mod_set_aligned(void *cbarg, cred_t *cr, mod_prop_info_t *pinfo,
152     const char *ifname, const void* pval, uint_t flags)
153 {
154         int     err;
155
156         if ((err = mod_set_uint32(cbarg, cr, pinfo, ifname, pval, flags)) != 0)
157                 return (err);
158
159         /* if required, align the value to multiple of 8 */
160         if (pinfo->prop_cur_uval & 0x7) {
161                 pinfo->prop_cur_uval &= ~0x7;
162                 pinfo->prop_cur_uval += 0x8;
163         }
164
165         return (0);
166 }
167
168 /*
169  * Retrieves property permission, default value, current value or possible
170  * values for those properties whose value type is uint32_t.
171  */
172 /* ARGSUSED */
173 int
174 mod_get_uint32(void *cbarg, mod_prop_info_t *pinfo, const char *ifname,
175     void *pval, uint_t psize, uint_t flags)
176 {
177         boolean_t       get_def = (flags & MOD_PROP_DEFAULT);
178         boolean_t       get_perm = (flags & MOD_PROP_PERM);
179         boolean_t       get_range = (flags & MOD_PROP_POSSIBLE);
180         size_t          nbytes;
181
182         bzero(pval, psize);
183         if (get_perm)
184                 nbytes = snprintf(pval, psize, "%u", prop_perm2const(pinfo));
185         else if (get_range)
186                 nbytes = snprintf(pval, psize, "%u-%u",
187                     pinfo->prop_min_uval, pinfo->prop_max_uval);
188         else if (get_def)
189                 nbytes = snprintf(pval, psize, "%u", pinfo->prop_def_uval);
190         else
191                 nbytes = snprintf(pval, psize, "%u", pinfo->prop_cur_uval);
192         if (nbytes >= psize)
193                 return (ENOBUFS);
```



```
194         return (0);
195 }
196
197 /*
198  * Implements /sbin/ndd -get /dev/ip ?, for all the modules. Needed for
199  * backward compatibility with /sbin/ndd.
200  */
201 /* ARGSUSED */
202 int
203 mod_get_allprop(void *cbarg, mod_prop_info_t *pinfo, const char *ifname,
204     void *val, uint_t psize, uint_t flags)
205 {
206         char            *pval = val;
207         mod_prop_info_t *ptbl, *prop;
208         ip_stack_t      *ipst;
209         tcp_stack_t     *tcps;
210         sctp_stack_t    *sctps;
211         dccp_stack_t    *dccps;
212 #endif /* ! codereview */
213         udp_stack_t     *us;
214         icmp_stack_t    *is;
215         uint_t          size;
216         size_t          nbytes = 0, tbytes = 0;
217
218         bzero(pval, psize);
219         size = psize;
220
221         switch (pinfo->mpi_proto) {
222         case MOD_PROTO_IP:
223         case MOD_PROTO_IPV4:
224         case MOD_PROTO_IPV6:
225                 ipst = (ip_stack_t *)cbarg;
226                 ptbl = ipst->ips_propinfo_tbl;
227                 break;
228         case MOD_PROTO_RAWIP:
229                 is = (icmp_stack_t *)cbarg;
230                 ptbl = is->is_propinfo_tbl;
231                 break;
232         case MOD_PROTO_TCP:
233                 tcps = (tcp_stack_t *)cbarg;
234                 ptbl = tcps->tcps_propinfo_tbl;
235                 break;
236         case MOD_PROTO_UDP:
237                 us = (udp_stack_t *)cbarg;
238                 ptbl = us->us_propinfo_tbl;
239                 break;
240         case MOD_PROTO_SCTP:
241                 sctps = (sctp_stack_t *)cbarg;
242                 ptbl = sctps->sctps_propinfo_tbl;
243                 break;
244         case MOD_PROTO_DCCP:
245                 dccps = (dccp_stack_t *)cbarg;
246                 ptbl = dccps->dccps_propinfo_tbl;
247                 break;
248 #endif /* ! codereview */
249         default:
250                 return (EINVAL);
251         }
252
253         for (prop = ptbl; prop->mpi_name != NULL; prop++) {
254                 if (prop->mpi_name[0] == ‘\0’ ||
255                     strcmp(prop->mpi_name, "?") == 0) {
256                         continue;
257                 }
258                 nbytes = snprintf(pval, size, "%s %d %d", prop->mpi_name,
259                     prop->mpi_proto, prop_perm2const(prop));
```

```
260                 size -= nbytes + 1;
261                 pval += nbytes + 1;
262                 tbytes += nbytes + 1;
263                 if (tbytes >= psize) {
264                         /* Buffer overflow, stop copying information */
265                         return (ENOBUFS);
266                 }
267         }
268         return (0);
269 }

271 /*
272  * Hold a lock while changing *_epriv_ports to prevent multiple
273  * threads from changing it at the same time.
274  */
275 /* ARGSUSED */
276 int
277 mod_set_extra_privports(void *cbarg, cred_t *cr, mod_prop_info_t *pinfo,
278     const char *ifname, const void* val, uint_t flags)
279 {
280         uint_t          proto = pinfo->mpi_proto;
281         tcp_stack_t     *tcps;
282         sctp_stack_t    *sctps;
283         udp_stack_t     *us;
284         unsigned long   new_value;
285         char            *end;
286         kmutex_t        *lock;
287         uint_t          i, nports;
288         in_port_t       *ports;
289         boolean_t       def = (flags & MOD_PROP_DEFAULT);
290         const char      *pval = val;

292         if (!def) {
293                 if (ddi_strtoul(pval, &end, 10, &new_value) != 0 ||
294                     *end != '\0') {
295                         return (EINVAL);
296                 }

298                 if (new_value < pinfo->prop_min_uval ||
299                     new_value > pinfo->prop_max_uval) {
300                         return (ERANGE);
301                 }
302         }

304         switch (proto) {
305         case MOD_PROTO_TCP:
306                 tcps = (tcp_stack_t *)cbarg;
307                 lock = &tcps->tcps_epriv_port_lock;
308                 ports = tcps->tcps_g_epriv_ports;
309                 nports = tcps->tcps_g_num_epriv_ports;
310                 break;
311         case MOD_PROTO_UDP:
312                 us = (udp_stack_t *)cbarg;
313                 lock = &us->us_epriv_port_lock;
314                 ports = us->us_epriv_ports;
315                 nports = us->us_num_epriv_ports;
316                 break;
317         case MOD_PROTO_SCTP:
318                 sctps = (sctp_stack_t *)cbarg;
319                 lock = &sctps->sctps_epriv_port_lock;
320                 ports = sctps->sctps_g_epriv_ports;
321                 nports = sctps->sctps_g_num_epriv_ports;
322                 break;
323         default:
324                 return (ENOTSUP);
325         }
```



```
327         mutex_enter(lock);

329         /* if MOD_PROP_DEFAULT is set then reset the ports list to default */
330         if (def) {
331                 for (i = 0; i < nports; i++)
332                         ports[i] = 0;
333                 ports[0] = ULP_DEF_EPRIV_PORT1;
334                 ports[1] = ULP_DEF_EPRIV_PORT2;
335                 mutex_exit(lock);
336                 return (0);
337         }

339         /* Check if the value is already in the list */
340         for (i = 0; i < nports; i++) {
341                 if (new_value == ports[i])
342                         break;
343         }

345         if (flags & MOD_PROP_REMOVE) {
346                 if (i == nports) {
347                         mutex_exit(lock);
348                         return (ESRCH);
349                 }
350                 /* Clear the value */
351                 ports[i] = 0;
352         } else if (flags & MOD_PROP_APPEND) {
353                 if (i != nports) {
354                         mutex_exit(lock);
355                         return (EEXIST);
356                 }

358                 /* Find an empty slot */
359                 for (i = 0; i < nports; i++) {
360                         if (ports[i] == 0)
361                                 break;
362                 }
363                 if (i == nports) {
364                         mutex_exit(lock);
365                         return (EOVERFLOW);
366                 }
367                 /* Set the new value */
368                 ports[i] = (in_port_t)new_value;
369         } else {
370                 /*
371                  * If the user used 'assignment' modifier.
372                  * For eg:
373                  *      # ipadm set-prop -p extra_priv_ports=3001 tcp
374                  *
375                  * We clear all the ports and then just add 3001.
376                  */
377                 ASSERT(flags == MOD_PROP_ACTIVE);
378                 for (i = 0; i < nports; i++)
379                         ports[i] = 0;
380                 ports[0] = (in_port_t)new_value;
381         }

383         mutex_exit(lock);
384         return (0);
385 }

387 /*
388  * Note: No locks are held when inspecting *_epriv_ports
389  * but instead the code relies on:
390  * - the fact that the address of the array and its size never changes
391  * - the atomic assignment of the elements of the array
```

```
 */
/* ARGSUSED */
int
mod_get_extra_privports(void *cbarg, mod_prop_info_t *pinfo, const char *ifname,
    void *val, uint_t psize, uint_t flags)
{
        uint_t          proto = pinfo->mpi_proto;
        tcp_stack_t     *tcps;
        sctp_stack_t    *sctps;
        udp_stack_t     *us;
        uint_t          i, nports, size;
        in_port_t       *ports;
        char            *pval = val;
        size_t          nbytes = 0, tbytes = 0;
        boolean_t       get_def = (flags & MOD_PROP_DEFAULT);
        boolean_t       get_perm = (flags & MOD_PROP_PERM);
        boolean_t       get_range = (flags & MOD_PROP_POSSIBLE);

        bzero(pval, psize);
        size = psize;

        if (get_def) {
                tbytes = snprintf(pval, psize, "%u,%u", ULP_DEF_EPRIV_PORT1,
                    ULP_DEF_EPRIV_PORT2);
                goto ret;
        } else if (get_perm) {
                tbytes = snprintf(pval, psize, "%u", MOD_PROP_PERM_RW);
                goto ret;
        }

        switch (proto) {
        case MOD_PROTO_TCP:
                tcps = (tcp_stack_t *)cbarg;
                ports = tcps->tcps_g_epriv_ports;
                nports = tcps->tcps_g_num_epriv_ports;
                break;
        case MOD_PROTO_UDP:
                us = (udp_stack_t *)cbarg;
                ports = us->us_epriv_ports;
                nports = us->us_num_epriv_ports;
                break;
        case MOD_PROTO_SCTP:
                sctps = (sctp_stack_t *)cbarg;
                ports = sctps->sctps_g_epriv_ports;
                nports = sctps->sctps_g_num_epriv_ports;
                break;
        default:
                return (ENOTSUP);
        }

        if (get_range) {
                tbytes = snprintf(pval, psize, "%u-%u", pinfo->prop_min_uval,
                    pinfo->prop_max_uval);
                goto ret;
        }

        for (i = 0; i < nports; i++) {
                if (ports[i] != 0) {
                        if (psize == size)
                                nbytes = snprintf(pval, size, "%u", ports[i]);
                        else
                                nbytes = snprintf(pval, size, ",%u", ports[i]);
                        size -= nbytes;
                        pval += nbytes;
                        tbytes += nbytes;
                        if (tbytes >= psize)
```



```
                                return (ENOBUFS);
                }
        }
        return (0);
ret:
        if (tbytes >= psize)
                return (ENOBUFS);
        return (0);
}
```

```
**********************************************************
    6181 Mon Jul  9 14:38:21 2012
new/usr/src/uts/common/inet/tunables.h
dccp: starting module template
**********************************************************
______unchanged_portion_omitted_

  59 #define MOD_PROP_VERSION        1

  61 /* permission flags for properties */
  62 #define MOD_PROP_PERM_READ      0x1
  63 #define MOD_PROP_PERM_WRITE     0x2
  64 #define MOD_PROP_PERM_RW        (MOD_PROP_PERM_READ|MOD_PROP_PERM_WRITE)

  66 /* mpr_flags values */
  67 #define MOD_PROP_ACTIVE         0x01    /* current value of the property */
  68 #define MOD_PROP_DEFAULT        0x02    /* default value of the property */
  69 #define MOD_PROP_POSSIBLE       0x04    /* possible values for the property */
  70 #define MOD_PROP_PERM           0x08    /* read/write permission for property */
  71 #define MOD_PROP_APPEND         0x10    /* append to multi-valued property */
  72 #define MOD_PROP_REMOVE         0x20    /* remove from multi-valued property */

  74 /* mpr_proto values */
  75 #define MOD_PROTO_NONE          0x00
  76 #define MOD_PROTO_IPV4          0x01    /* property is applicable to IPV4 */
  77 #define MOD_PROTO_IPV6          0x02    /* property is applicable to IPV6 */
  78 #define MOD_PROTO_RAWIP         0x04    /* property is applicable to ICMP */
  79 #define MOD_PROTO_TCP           0x08    /* property is applicable to TCP */
  80 #define MOD_PROTO_UDP           0x10    /* property is applicable to UDP */
  81 #define MOD_PROTO_SCTP          0x20    /* property is applicable to SCTP */
  82 #define MOD_PROTO_DCCP          0x40    /* property is applicable to DCCP */
  83 #endif /* ! codereview */

  85 /* property is applicable to both IPV[4|6] */
  86 #define MOD_PROTO_IP            (MOD_PROTO_IPV4|MOD_PROTO_IPV6)

  88 #ifdef  _KERNEL

  90 typedef struct mod_prop_info_s mod_prop_info_t;

  92 /* set/get property callback functions */
  93 typedef int     mod_prop_setf_t(void *, cred_t *, mod_prop_info_t *,
  94                     const char *, const void *, uint_t);
  95 typedef int     mod_prop_getf_t(void *, mod_prop_info_t *, const char *,
  96                     void *val, uint_t, uint_t);

  98 typedef struct mod_propval_uint32_s {
  99         uint32_t        mod_propval_umin;
 100         uint32_t        mod_propval_umax;
 101         uint32_t        mod_propval_ucur;
 102 } mod_propval_uint32_t;

 104 /*
 105  * protocol property information
 106  */
 107 struct mod_prop_info_s {
 108         char                    *mpi_name;      /* property name */
 109         uint_t                  mpi_proto;      /* property protocol */
 110         mod_prop_setf_t         *mpi_setf;      /* sets the property value */
 111         mod_prop_getf_t         *mpi_getf;      /* gets the property value */
 112         /*
 113          * Holds the current value of the property. Whenever applicable
 114          * holds the min/max value too.
 115          */
 116         union {
 117                 mod_propval_uint32_t    mpi_uval;
```



```
 118                 boolean_t               mpi_bval;
 119                 uint64_t                _pad[2];
 120         } u;
 121         /*
 122          * Holds the default value of the property, that is value of
 123          * the property at boot time.
 124          */
 125         union {
 126                 uint32_t        mpi_def_uval;
 127                 boolean_t       mpi_def_bval;
 128         } u_def;
 129 };

 131 /* shortcuts to access current/default values */
 132 #define prop_min_uval   u.mpi_uval.mod_propval_umin
 133 #define prop_max_uval   u.mpi_uval.mod_propval_umax
 134 #define prop_cur_uval   u.mpi_uval.mod_propval_ucur
 135 #define prop_cur_bval   u.mpi_bval
 136 #define prop_def_uval   u_def.mpi_def_uval
 137 #define prop_def_bval   u_def.mpi_def_bval

 139 #define MS              1L
 140 #define SECONDS         (1000 * MS)
 141 #define MINUTES         (60 * SECONDS)
 142 #define HOURS           (60 * MINUTES)
 143 #define DAYS            (24 * HOURS)

 145 #define MB              (1024 * 1024)

 147 /* Largest TCP/UDP/SCTP port number */
 148 #define ULP_MAX_PORT    (64 * 1024 - 1)

 150 /* extra privilege ports for upper layer protocols, tcp, sctp and udp */
 151 #define ULP_DEF_EPRIV_PORT1     2049
 152 #define ULP_DEF_EPRIV_PORT2     4045

 154 /* generic function to set/get global module properties */
 155 extern mod_prop_setf_t  mod_set_boolean, mod_set_uint32,
 156                         mod_set_aligned, mod_set_extra_privports;

 158 extern mod_prop_getf_t  mod_get_boolean, mod_get_uint32,
 159                         mod_get_allprop, mod_get_extra_privports;

 161 extern int mod_uint32_value(const void *, mod_prop_info_t *, uint_t,
 162     unsigned long *);

 164 #endif  /* _KERNEL */

 166 /*
 167  * End-system model definitions that include the weak/strong end-system
 168  * definitions in RFC 1122, Section 3.3.4.5. IP_WEAK_ES and IP_STRONG_ES
 169  * conform to the corresponding  RFC 1122 definitions. The IP_SRC_PRI_ES
 170  * hostmodel is similar to IP_WEAK_ES with one additional enhancement: for
 171  * a packet with source S2, destination D2, the route selection algorithm
 172  * will first attempt to find a route for the destination that goes out
 173  * through an interface where S2 is configured and marked UP.  If such
 174  * a route cannot be found, then the best-matching route for D2 will be
 175  * selected, ignoring any mismatches between S2 and the interface addresses
 176  * on the outgoing interface implied by the route.
 177  */
 178 typedef enum {
 179         IP_WEAK_ES = 0,
 180         IP_SRC_PRI_ES,
 181         IP_STRONG_ES,
 182         IP_MAXVAL_ES
 183 } ip_hostmodel_t;
```

```
185 #ifdef  __cplusplus
186 }
187 #endif

189 #endif  /* _INET_TUNABLES_H */
```

```
**********************************************************
    1543 Mon Jul  9 14:38:21 2012
new/usr/src/uts/common/netinet/Makefile
dccp: snoop, build system fixes
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
#ident  "%Z%%M% %I%     %E% SMI"
#
# Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# uts/common/netinet/Makefile
#
# include global definitions
include ../../../Makefile.master

HDRS=   arp.h dccp.h dhcp.h dhcp6.h icmp6.h icmp_var.h if_ether.h igmp.h \
        igmp_var.h in.h inetutil.h in_pcb.h in_systm.h in_var.h ip.h ip6.h \
        ip_icmp.h ip_mroute.h ip_var.h pim.h sctp.h tcp.h tcp_debug.h \
        tcp_fsm.h tcp_seq.h tcp_timer.h tcp_var.h tcpip.h udp.h udp_var.h
HDRS=   arp.h dhcp.h dhcp6.h icmp6.h icmp_var.h if_ether.h igmp.h igmp_var.h \
        in.h inetutil.h in_pcb.h in_systm.h in_var.h ip.h ip6.h ip_icmp.h \
        ip_mroute.h ip_var.h pim.h sctp.h tcp.h tcp_debug.h tcp_fsm.h \
        tcp_seq.h tcp_timer.h tcp_var.h tcpip.h udp.h udp_var.h

ROOTDIRS= $(ROOT)/usr/include/netinet

ROOTHDRS= $(HDRS:%=$(ROOT)/usr/include/netinet/%)

CHECKHDRS= $(HDRS:%.h=%.check)

$(ROOTDIRS)/%: %
        $(INS.file)

.KEEP_STATE:

.PARALLEL: $(CHECKHDRS)

install_h: $(ROOTDIRS) $(ROOTHDRS)

$(ROOTDIRS):
        $(INS.dir)

check:  $(CHECKHDRS)
```

```
**********************************************************
    1380 Mon Jul  9 14:38:22 2012
new/usr/src/uts/common/netinet/dccp.h
dccp: starting module template
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2012 David Hoeppner.  All rights reserved.
 */

#ifndef _NETINET_DCCP_H
#define _NETINET_DCCP_H

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * DCCP header
 */
struct dccphdr {
        uint16_t        dh_sport;
        uint16_t        dh_dport;
        uint8_t         dh_offset;
#ifdef _BIT_FIELDS_LTOH
        uint8_t         dha_ccval:4,
                        dha_cscov:4;
#else
        uint8_t         dha_cscov:4,
                        dha_ccval:4;
#endif
        uint16_t        dha_sum;
#ifdef _BIT_FIELDS_LTOH
        uint8_t         dha_x:1,
                        dha_type:4,
                        dha_reserved:3;
#else
        uint8_t         dha_reserved:3,
                        dha_type:4,
                        dha_x:1;
#endif
        uint8_t         dha_res_seq;
        uint16_t        dha_seq;
};

/*
 * DCCP states
 */
#define DCCPS_CLOSED    -5
#define DCCPS_IDLE      -5
#define DCCPS_BOUND     -4
#define DCCPS_LISTEN    -3
#define DCCPS_REQUEST   -2
#define DCCPS_REQUEST_SENT      -2
#define DCCPS_REQUEST_RCVD      -2
#define DCCPS_RESPOND   -1
```



```
#define DCCPS_ACK_RCVD  -1
#define DCCPS_PARTOPEN  0
#define DCCPS_ACK_SENT          1
#define DCCPS_ESTABLISHED       2


#ifdef  __cplusplus
}
#endif

#endif  /* _NETINET_DCCP_H */
#endif /* ! codereview */
```

```
**********************************************************
   44002 Mon Jul  9 14:38:22 2012
new/usr/src/uts/common/netinet/in.h
dccp: starting module template
**********************************************************
______unchanged_portion_omitted_
 128 #define s6_addr         _S6_un._S6_u8

 130 #ifdef _KERNEL
 131 #define s6_addr8        _S6_un._S6_u8
 132 #define s6_addr32       _S6_un._S6_u32
 133 #endif

 135 typedef struct in6_addr in6_addr_t;

 137 #endif /* !defined(_XPG4_2) || defined(_XPG6) || defined(__EXTENSIONS__) */

 139 #ifndef _SA_FAMILY_T
 140 #define _SA_FAMILY_T
 141 typedef uint16_t        sa_family_t;
 142 #endif

 144 /*
 145  * Protocols
 146  *
 147  * Some of these constant names are copied for the DTrace IP provider in
 148  * usr/src/lib/libdtrace/common/{ip.d.in, ip.sed.in}, which should be kept
 149  * in sync.
 150  */
 151 #define IPPROTO_IP              0               /* dummy for IP */
 152 #define IPPROTO_HOPOPTS         0               /* Hop by hop header for IPv6 */
 153 #define IPPROTO_ICMP            1               /* control message protocol */
 154 #define IPPROTO_IGMP            2               /* group control protocol */
 155 #define IPPROTO_GGP             3               /* gateway^2 (deprecated) */
 156 #define IPPROTO_ENCAP           4               /* IP in IP encapsulation */
 157 #define IPPROTO_TCP             6               /* tcp */
 158 #define IPPROTO_EGP             8               /* exterior gateway protocol */
 159 #define IPPROTO_PUP             12              /* pup */
 160 #define IPPROTO_UDP             17              /* user datagram protocol */
 161 #define IPPROTO_IDP             22              /* xns idp */
 162 #define IPPROTO_DCCP            33              /* DCCP */
 163 #endif /* ! codereview */
 164 #define IPPROTO_IPV6            41              /* IPv6 encapsulated in IP */
 165 #define IPPROTO_ROUTING         43              /* Routing header for IPv6 */
 166 #define IPPROTO_FRAGMENT        44              /* Fragment header for IPv6 */
 167 #define IPPROTO_RSVP            46              /* rsvp */
 168 #define IPPROTO_ESP             50              /* IPsec Encap. Sec. Payload */
 169 #define IPPROTO_AH              51              /* IPsec Authentication Hdr. */
 170 #define IPPROTO_ICMPV6          58              /* ICMP for IPv6 */
 171 #define IPPROTO_NONE            59              /* No next header for IPv6 */
 172 #define IPPROTO_DSTOPTS         60              /* Destination options */
 173 #define IPPROTO_HELLO           63              /* "hello" routing protocol */
 174 #define IPPROTO_ND              77              /* UNOFFICIAL net disk proto */
 175 #define IPPROTO_EON             80              /* ISO clnp */
 176 #define IPPROTO_OSPF            89              /* OSPF */
 177 #define IPPROTO_PIM             103             /* PIM routing protocol */
 178 #define IPPROTO_SCTP            132             /* Stream Control */
 179                                                 /* Transmission Protocol */

 181 #define IPPROTO_RAW             255             /* raw IP packet */
 182 #define IPPROTO_MAX             256

 184 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
 185 #define PROTO_SDP               257             /* Sockets Direct Protocol */
 186 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */
```



```
 188 /*
 189  * Port/socket numbers: network standard functions
 190  *
 191  * Entries should exist here for each port number compiled into an ON
 192  * component, such as snoop.
 193  */
 194 #define IPPORT_ECHO             7
 195 #define IPPORT_DISCARD          9
 196 #define IPPORT_SYSTAT           11
 197 #define IPPORT_DAYTIME          13
 198 #define IPPORT_NETSTAT          15
 199 #define IPPORT_CHARGEN          19
 200 #define IPPORT_FTP              21
 201 #define IPPORT_TELNET           23
 202 #define IPPORT_SMTP             25
 203 #define IPPORT_TIMESERVER       37
 204 #define IPPORT_NAMESERVER       42
 205 #define IPPORT_WHOIS            43
 206 #define IPPORT_DOMAIN           53
 207 #define IPPORT_MDNS             5353
 208 #define IPPORT_MTP              57

 210 /*
 211  * Port/socket numbers: host specific functions
 212  */
 213 #define IPPORT_BOOTPS           67
 214 #define IPPORT_BOOTPC           68
 215 #define IPPORT_TFTP             69
 216 #define IPPORT_RJE              77
 217 #define IPPORT_FINGER           79
 218 #define IPPORT_HTTP             80
 219 #define IPPORT_HTTP_ALT         8080
 220 #define IPPORT_TTYLINK          87
 221 #define IPPORT_SUPDUP           95
 222 #define IPPORT_NTP              123
 223 #define IPPORT_NETBIOS_NS       137
 224 #define IPPORT_NETBIOS_DGM      138
 225 #define IPPORT_NETBIOS_SSN      139
 226 #define IPPORT_LDAP             389
 227 #define IPPORT_SLP              427
 228 #define IPPORT_MIP              434
 229 #define IPPORT_SMB              445             /* a.k.a. microsoft-ds */

 231 /*
 232  * Internet Key Exchange (IKE) ports
 233  */
 234 #define IPPORT_IKE              500
 235 #define IPPORT_IKE_NATT         4500

 237 /*
 238  * UNIX TCP sockets
 239  */
 240 #define IPPORT_EXECSERVER       512
 241 #define IPPORT_LOGINSERVER      513
 242 #define IPPORT_CMDSERVER        514
 243 #define IPPORT_PRINTER          515
 244 #define IPPORT_EFSSERVER        520

 246 /*
 247  * UNIX UDP sockets
 248  */
 249 #define IPPORT_BIFFUDP          512
 250 #define IPPORT_WHOSERVER        513
 251 #define IPPORT_SYSLOG           514
 252 #define IPPORT_TALK             517
 253 #define IPPORT_ROUTESERVER      520
```

```
254 #define IPPORT_RIPNG            521

256 /*
257  * DHCPv6 UDP ports
258  */
259 #define IPPORT_DHCPV6C          546
260 #define IPPORT_DHCPV6S          547

262 #define IPPORT_SOCKS            1080

264 /*
265  * Ports < IPPORT_RESERVED are reserved for
266  * privileged processes (e.g. root).
267  * Ports > IPPORT_USERRESERVED are reserved
268  * for servers, not necessarily privileged.
269  */
270 #define IPPORT_RESERVED         1024
271 #define IPPORT_USERRESERVED     5000

273 /*
274  * Link numbers
275  */
276 #define IMPLINK_IP              155
277 #define IMPLINK_LOWEXPER        156
278 #define IMPLINK_HIGHEXPER       158

280 /*
281  * IPv4 Internet address
282  *      This definition contains obsolete fields for compatibility
283  *      with SunOS 3.x and 4.2bsd.  The presence of subnets renders
284  *      divisions into fixed fields misleading at best.  New code
285  *      should use only the s_addr field.
286  */

288 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
289 #define _S_un_b S_un_b
290 #define _S_un_w S_un_w
291 #define _S_addr S_addr
292 #define _S_un   S_un
293 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

295 struct in_addr {
296         union {
297                 struct { uint8_t s_b1, s_b2, s_b3, s_b4; } _S_un_b;
298                 struct { uint16_t s_w1, s_w2; } _S_un_w;
299 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
300                 uint32_t _S_addr;
301 #else
302                 in_addr_t _S_addr;
303 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */
304         } _S_un;
305 #define s_addr  _S_un._S_addr           /* should be used for all code */
306 #define s_host  _S_un._S_un_b.s_b2      /* OBSOLETE: host on imp */
307 #define s_net   _S_un._S_un_b.s_b1      /* OBSOLETE: network */
308 #define s_imp   _S_un._S_un_w.s_w2      /* OBSOLETE: imp */
309 #define s_impno _S_un._S_un_b.s_b4      /* OBSOLETE: imp # */
310 #define s_lh    _S_un._S_un_b.s_b3      /* OBSOLETE: logical host */
311 };

313 /*
314  * Definitions of bits in internet address integers.
315  * On subnets, the decomposition of addresses to host and net parts
316  * is done according to subnet mask, not the masks here.
317  *
318  * Note that with the introduction of CIDR, IN_CLASSA, IN_CLASSB,
319  * IN_CLASSC, IN_CLASSD and IN_CLASSE macros have become "de-facto
```



```
320  * obsolete". IN_MULTICAST macro should be used to test if a address
321  * is a multicast address.
322  */
323 #define IN_CLASSA(i)            (((i) & 0x80000000U) == 0)
324 #define IN_CLASSA_NET           0xff000000U
325 #define IN_CLASSA_NSHIFT        24
326 #define IN_CLASSA_HOST          0x00ffffffU
327 #define IN_CLASSA_MAX           128

329 #define IN_CLASSB(i)            (((i) & 0xc0000000U) == 0x80000000U)
330 #define IN_CLASSB_NET           0xffff0000U
331 #define IN_CLASSB_NSHIFT        16
332 #define IN_CLASSB_HOST          0x0000ffffU
333 #define IN_CLASSB_MAX           65536

335 #define IN_CLASSC(i)            (((i) & 0xe0000000U) == 0xc0000000U)
336 #define IN_CLASSC_NET           0xffffff00U
337 #define IN_CLASSC_NSHIFT        8
338 #define IN_CLASSC_HOST          0x000000ffU

340 #define IN_CLASSD(i)            (((i) & 0xf0000000U) == 0xe0000000U)
341 #define IN_CLASSD_NET           0xf0000000U     /* These aren't really  */
342 #define IN_CLASSD_NSHIFT        28              /* net and host fields, but */
343 #define IN_CLASSD_HOST          0x0fffffffU     /* routing needn't know */

345 #define IN_CLASSE(i)            (((i) & 0xf0000000U) == 0xf0000000U)
346 #define IN_CLASSE_NET           0xffffffffU

348 #define IN_MULTICAST(i)         IN_CLASSD(i)

350 /*
351  * We have removed CLASS E checks from the kernel
352  * But we preserve these defines for userland in order
353  * to avoid compile  breakage of some 3rd party piece of software
354  */
355 #ifndef _KERNEL
356 #define IN_EXPERIMENTAL(i)      (((i) & 0xe0000000U) == 0xe0000000U)
357 #define IN_BADCLASS(i)          (((i) & 0xf0000000U) == 0xf0000000U)
358 #endif

360 #define INADDR_ANY              0x00000000U
361 #define INADDR_LOOPBACK         0x7F000001U
362 #define INADDR_BROADCAST        0xffffffffU     /* must be masked */
363 #define INADDR_NONE             0xffffffffU

365 #define INADDR_UNSPEC_GROUP     0xe0000000U     /* 224.0.0.0   */
366 #define INADDR_ALLHOSTS_GROUP   0xe0000001U     /* 224.0.0.1   */
367 #define INADDR_ALLRTRS_GROUP    0xe0000002U     /* 224.0.0.2   */
368 #define INADDR_ALLRPTS_GROUP    0xe0000016U     /* 224.0.0.22, IGMPv3 */
369 #define INADDR_MAX_LOCAL_GROUP  0xe00000ffU     /* 224.0.0.255 */

371 /* Scoped IPv4 prefixes (in host byte-order) */
372 #define IN_AUTOCONF_NET         0xa9fe0000U     /* 169.254/16 */
373 #define IN_AUTOCONF_MASK        0xffff0000U
374 #define IN_PRIVATE8_NET         0x0a000000U     /* 10/8 */
375 #define IN_PRIVATE8_MASK        0xff000000U
376 #define IN_PRIVATE12_NET        0xac100000U     /* 172.16/12 */
377 #define IN_PRIVATE12_MASK       0xfff00000U
378 #define IN_PRIVATE16_NET        0xc0a80000U     /* 192.168/16 */
379 #define IN_PRIVATE16_MASK       0xffff0000U

381 /* RFC 3927 IPv4 link local address (i in host byte-order) */
382 #define IN_LINKLOCAL(i)         (((i) & IN_AUTOCONF_MASK) == IN_AUTOCONF_NET)

384 /* Well known 6to4 Relay Router Anycast address defined in RFC 3068 */
385 #if !defined(_XPG4_2) || !defined(__EXTENSIONS__)
```

```
386 #define INADDR_6TO4RRANYCAST    0xc0586301U     /* 192.88.99.1 */
387 #endif  /* !defined(_XPG4_2) || !defined(__EXTENSIONS__) */

389 #define IN_LOOPBACKNET          127                     /* official! */

391 /*
392  * Define a macro to stuff the loopback address into an Internet address
393  */
394 #if !defined(_XPG4_2) || !defined(__EXTENSIONS__)
395 #define IN_SET_LOOPBACK_ADDR(a) \
396         { (a)->sin_addr.s_addr  = htonl(INADDR_LOOPBACK); \
397         (a)->sin_family = AF_INET; }
398 #endif /* !defined(_XPG4_2) || !defined(__EXTENSIONS__) */

400 /*
401  * IPv4 Socket address.
402  */
403 struct sockaddr_in {
404         sa_family_t     sin_family;
405         in_port_t       sin_port;
406         struct  in_addr sin_addr;
407 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
408         char            sin_zero[8];
409 #else
410         unsigned char   sin_zero[8];
411 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */
412 };

414 #if !defined(_XPG4_2) || defined(_XPG6) || defined(__EXTENSIONS__)
415 /*
416  * IPv6 socket address.
417  */
418 struct sockaddr_in6 {
419         sa_family_t     sin6_family;
420         in_port_t       sin6_port;
421         uint32_t        sin6_flowinfo;
422         struct in6_addr sin6_addr;
423         uint32_t        sin6_scope_id;  /* Depends on scope of sin6_addr */
424         uint32_t        __sin6_src_id;  /* Impl. specific - UDP replies */
425 };

427 /*
428  * Macros for accessing the traffic class and flow label fields from
429  * sin6_flowinfo.
430  * These are designed to be applied to a 32-bit value.
431  */
432 #ifdef _BIG_ENDIAN

434 /* masks */
435 #define IPV6_FLOWINFO_FLOWLABEL                 0x000fffffU
436 #define IPV6_FLOWINFO_TCLASS                    0x0ff00000U

438 #else /* _BIG_ENDIAN */

440 /* masks */
441 #define IPV6_FLOWINFO_FLOWLABEL                 0xffff0f00U
442 #define IPV6_FLOWINFO_TCLASS                    0x0000f00fU

444 #endif  /* _BIG_ENDIAN */

446 /*
447  * Note: Macros IN6ADDR_ANY_INIT and IN6ADDR_LOOPBACK_INIT are for
448  * use as RHS of Static initializers of "struct in6_addr" (or in6_addr_t)
449  * only. They need to be different for User/Kernel versions because union
450  * component data structure is defined differently (it is identical at
451  * binary representation level).
```



```
452  *
453  * const struct in6_addr IN6ADDR_ANY_INIT;
454  * const struct in6_addr IN6ADDR_LOOPBACK_INIT;
455  */


458 #ifdef _KERNEL
459 #define IN6ADDR_ANY_INIT                { 0, 0, 0, 0 }

461 #ifdef _BIG_ENDIAN
462 #define IN6ADDR_LOOPBACK_INIT           { 0, 0, 0, 0x00000001U }
463 #else /* _BIG_ENDIAN */
464 #define IN6ADDR_LOOPBACK_INIT           { 0, 0, 0, 0x01000000U }
465 #endif /* _BIG_ENDIAN */

467 #else

469 #define IN6ADDR_ANY_INIT            {   0, 0, 0, 0,     \
470                                         0, 0, 0, 0,     \
471                                         0, 0, 0, 0,     \
472                                         0, 0, 0, 0 }

474 #define IN6ADDR_LOOPBACK_INIT       {   0, 0, 0, 0,     \
475                                         0, 0, 0, 0,     \
476                                         0, 0, 0, 0,     \
477                                         0, 0, 0, 0x1U }
478 #endif /* _KERNEL */

480 /*
481  * RFC 2553 specifies the following macros. Their type is defined
482  * as "int" in the RFC but they only have boolean significance
483  * (zero or non-zero). For the purposes of our comment notation,
484  * we assume a hypothetical type "bool" defined as follows to
485  * write the prototypes assumed for macros in our comments better.
486  *
487  * typedef int bool;
488  */

490 /*
491  * IN6 macros used to test for special IPv6 addresses
492  * (Mostly from spec)
493  *
494  * bool  IN6_IS_ADDR_UNSPECIFIED (const struct in6_addr *);
495  * bool  IN6_IS_ADDR_LOOPBACK    (const struct in6_addr *);
496  * bool  IN6_IS_ADDR_MULTICAST   (const struct in6_addr *);
497  * bool  IN6_IS_ADDR_LINKLOCAL   (const struct in6_addr *);
498  * bool  IN6_IS_ADDR_SITELOCAL   (const struct in6_addr *);
499  * bool  IN6_IS_ADDR_V4MAPPED    (const struct in6_addr *);
500  * bool  IN6_IS_ADDR_V4MAPPED_ANY(const struct in6_addr *); -- Not from RFC2553
501  * bool  IN6_IS_ADDR_V4COMPAT    (const struct in6_addr *);
502  * bool  IN6_IS_ADDR_MC_RESERVED (const struct in6_addr *); -- Not from RFC2553
503  * bool  IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *);
504  * bool  IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *);
505  * bool  IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *);
506  * bool  IN6_IS_ADDR_MC_ORGLOCAL (const struct in6_addr *);
507  * bool  IN6_IS_ADDR_MC_GLOBAL   (const struct in6_addr *);
508  * bool  IN6_IS_ADDR_6TO4        (const struct in6_addr *); -- Not from RFC2553
509  * bool  IN6_ARE_6TO4_PREFIX_EQUAL(const struct in6_addr *,
510  *           const struct in6_addr *);                      -- Not from RFC2553
511  * bool  IN6_IS_ADDR_LINKSCOPE   (const struct in6addr  *); -- Not from RFC2553
512  */

514 #define IN6_IS_ADDR_UNSPECIFIED(addr) \
515         (((addr)->_S6_un._S6_u32[3] == 0) && \
516         ((addr)->_S6_un._S6_u32[2] == 0) && \
517         ((addr)->_S6_un._S6_u32[1] == 0) && \
```

```
518         ((addr)->_S6_un._S6_u32[0] == 0))

520 #ifdef _BIG_ENDIAN
521 #define IN6_IS_ADDR_LOOPBACK(addr) \
522         (((addr)->_S6_un._S6_u32[3] == 0x00000001) && \
523         ((addr)->_S6_un._S6_u32[2] == 0) && \
524         ((addr)->_S6_un._S6_u32[1] == 0) && \
525         ((addr)->_S6_un._S6_u32[0] == 0))
526 #else /* _BIG_ENDIAN */
527 #define IN6_IS_ADDR_LOOPBACK(addr) \
528         (((addr)->_S6_un._S6_u32[3] == 0x01000000) && \
529         ((addr)->_S6_un._S6_u32[2] == 0) && \
530         ((addr)->_S6_un._S6_u32[1] == 0) && \
531         ((addr)->_S6_un._S6_u32[0] == 0))
532 #endif /* _BIG_ENDIAN */

534 #ifdef _BIG_ENDIAN
535 #define IN6_IS_ADDR_MULTICAST(addr) \
536         (((addr)->_S6_un._S6_u32[0] & 0xff000000) == 0xff000000)
537 #else /* _BIG_ENDIAN */
538 #define IN6_IS_ADDR_MULTICAST(addr) \
539         (((addr)->_S6_un._S6_u32[0] & 0x000000ff) == 0x000000ff)
540 #endif /* _BIG_ENDIAN */

542 #ifdef _BIG_ENDIAN
543 #define IN6_IS_ADDR_LINKLOCAL(addr) \
544         (((addr)->_S6_un._S6_u32[0] & 0xffc00000) == 0xfe800000)
545 #else /* _BIG_ENDIAN */
546 #define IN6_IS_ADDR_LINKLOCAL(addr) \
547         (((addr)->_S6_un._S6_u32[0] & 0x0000c0ff) == 0x000080fe)
548 #endif /* _BIG_ENDIAN */

550 #ifdef _BIG_ENDIAN
551 #define IN6_IS_ADDR_SITELOCAL(addr) \
552         (((addr)->_S6_un._S6_u32[0] & 0xffc00000) == 0xfec00000)
553 #else /* _BIG_ENDIAN */
554 #define IN6_IS_ADDR_SITELOCAL(addr) \
555         (((addr)->_S6_un._S6_u32[0] & 0x0000c0ff) == 0x0000c0fe)
556 #endif /* _BIG_ENDIAN */

558 #ifdef _BIG_ENDIAN
559 #define IN6_IS_ADDR_V4MAPPED(addr) \
560         (((addr)->_S6_un._S6_u32[2] == 0x0000ffff) && \
561         ((addr)->_S6_un._S6_u32[1] == 0) && \
562         ((addr)->_S6_un._S6_u32[0] == 0))
563 #else  /* _BIG_ENDIAN */
564 #define IN6_IS_ADDR_V4MAPPED(addr) \
565         (((addr)->_S6_un._S6_u32[2] == 0xffff0000U) && \
566         ((addr)->_S6_un._S6_u32[1] == 0) && \
567         ((addr)->_S6_un._S6_u32[0] == 0))
568 #endif /* _BIG_ENDIAN */

570 /*
571  * IN6_IS_ADDR_V4MAPPED - A IPv4 mapped INADDR_ANY
572  * Note: This macro is currently NOT defined in RFC2553 specification
573  * and not a standard macro that portable applications should use.
574  */
575 #ifdef _BIG_ENDIAN
576 #define IN6_IS_ADDR_V4MAPPED_ANY(addr) \
577         (((addr)->_S6_un._S6_u32[3] == 0) && \
578         ((addr)->_S6_un._S6_u32[2] == 0x0000ffff) && \
579         ((addr)->_S6_un._S6_u32[1] == 0) && \
580         ((addr)->_S6_un._S6_u32[0] == 0))
581 #else  /* _BIG_ENDIAN */
582 #define IN6_IS_ADDR_V4MAPPED_ANY(addr) \
583         (((addr)->_S6_un._S6_u32[3] == 0) && \
```



```
584         ((addr)->_S6_un._S6_u32[2] == 0xffff0000U) && \
585         ((addr)->_S6_un._S6_u32[1] == 0) && \
586         ((addr)->_S6_un._S6_u32[0] == 0))
587 #endif /* _BIG_ENDIAN */

589 /* Exclude loopback and unspecified address */
590 #ifdef _BIG_ENDIAN
591 #define IN6_IS_ADDR_V4COMPAT(addr) \
592         (((addr)->_S6_un._S6_u32[2] == 0) && \
593         ((addr)->_S6_un._S6_u32[1] == 0) && \
594         ((addr)->_S6_un._S6_u32[0] == 0) && \
595         !((addr)->_S6_un._S6_u32[3] == 0) && \
596         !((addr)->_S6_un._S6_u32[3] == 0x00000001))

598 #else /* _BIG_ENDIAN */
599 #define IN6_IS_ADDR_V4COMPAT(addr) \
600         (((addr)->_S6_un._S6_u32[2] == 0) && \
601         ((addr)->_S6_un._S6_u32[1] == 0) && \
602         ((addr)->_S6_un._S6_u32[0] == 0) && \
603         !((addr)->_S6_un._S6_u32[3] == 0) && \
604         !((addr)->_S6_un._S6_u32[3] == 0x01000000))
605 #endif /* _BIG_ENDIAN */

607 /*
608  * Note:
609  * IN6_IS_ADDR_MC_RESERVED macro is currently NOT defined in RFC2553
610  * specification and not a standard macro that portable applications
611  * should use.
612  */
613 #ifdef _BIG_ENDIAN
614 #define IN6_IS_ADDR_MC_RESERVED(addr) \
615         (((addr)->_S6_un._S6_u32[0] & 0xff0f0000) == 0xff000000)

617 #else  /* _BIG_ENDIAN */
618 #define IN6_IS_ADDR_MC_RESERVED(addr) \
619         (((addr)->_S6_un._S6_u32[0] & 0x00000fff) == 0x000000ff)
620 #endif /* _BIG_ENDIAN */

622 #ifdef _BIG_ENDIAN
623 #define IN6_IS_ADDR_MC_NODELOCAL(addr) \
624         (((addr)->_S6_un._S6_u32[0] & 0xff0f0000) == 0xff010000)
625 #else  /* _BIG_ENDIAN */
626 #define IN6_IS_ADDR_MC_NODELOCAL(addr) \
627         (((addr)->_S6_un._S6_u32[0] & 0x00000fff) == 0x000001ff)
628 #endif /* _BIG_ENDIAN */

630 #ifdef _BIG_ENDIAN
631 #define IN6_IS_ADDR_MC_LINKLOCAL(addr) \
632         (((addr)->_S6_un._S6_u32[0] & 0xff0f0000) == 0xff020000)
633 #else  /* _BIG_ENDIAN */
634 #define IN6_IS_ADDR_MC_LINKLOCAL(addr) \
635         (((addr)->_S6_un._S6_u32[0] & 0x00000fff) == 0x000002ff)
636 #endif /* _BIG_ENDIAN */

638 #ifdef _BIG_ENDIAN
639 #define IN6_IS_ADDR_MC_SITELOCAL(addr) \
640         (((addr)->_S6_un._S6_u32[0] & 0xff0f0000) == 0xff050000)
641 #else  /* _BIG_ENDIAN */
642 #define IN6_IS_ADDR_MC_SITELOCAL(addr) \
643         (((addr)->_S6_un._S6_u32[0] & 0x00000fff) == 0x000005ff)
644 #endif /* _BIG_ENDIAN */

646 #ifdef _BIG_ENDIAN
647 #define IN6_IS_ADDR_MC_ORGLOCAL(addr) \
648         (((addr)->_S6_un._S6_u32[0] & 0xff0f0000) == 0xff080000)
649 #else  /* _BIG_ENDIAN */
```

```
650 #define IN6_IS_ADDR_MC_ORGLOCAL(addr) \
651         (((addr)->_S6_un._S6_u32[0] & 0x00000fff) == 0x000008ff)
652 #endif /* _BIG_ENDIAN */

654 #ifdef _BIG_ENDIAN
655 #define IN6_IS_ADDR_MC_GLOBAL(addr) \
656         (((addr)->_S6_un._S6_u32[0] & 0xff0f0000) == 0xff0e0000)
657 #else /* _BIG_ENDIAN */
658 #define IN6_IS_ADDR_MC_GLOBAL(addr) \
659         (((addr)->_S6_un._S6_u32[0] & 0x00000fff) == 0x00000eff)
660 #endif /* _BIG_ENDIAN */

662 /*
663  * The IN6_IS_ADDR_MC_SOLICITEDNODE macro is not defined in any standard or
664  * RFC, and shouldn't be used by portable applications.  It is used to see
665  * if an address is a solicited-node multicast address, which is prefixed
666  * with ff02:0:0:0:0:1:ff00::/104.
667  */
668 #ifdef _BIG_ENDIAN
669 #define IN6_IS_ADDR_MC_SOLICITEDNODE(addr)                      \
670         (((addr)->_S6_un._S6_u32[0] == 0xff020000) &&           \
671         ((addr)->_S6_un._S6_u32[1] == 0x00000000) &&            \
672         ((addr)->_S6_un._S6_u32[2] == 0x00000001) &&            \
673         (((addr)->_S6_un._S6_u32[3] & 0xff000000) == 0xff000000))
674 #else
675 #define IN6_IS_ADDR_MC_SOLICITEDNODE(addr)                      \
676         (((addr)->_S6_un._S6_u32[0] == 0x000002ff) &&           \
677         ((addr)->_S6_un._S6_u32[1] == 0x00000000) &&            \
678         ((addr)->_S6_un._S6_u32[2] == 0x01000000) &&            \
679         (((addr)->_S6_un._S6_u32[3] & 0x000000ff) == 0x000000ff))
680 #endif

682 /*
683  * Macros to a) test for 6to4 IPv6 address, and b) to test if two
684  * 6to4 addresses have the same /48 prefix, and, hence, are from the
685  * same 6to4 site.
686  */

688 #ifdef _BIG_ENDIAN
689 #define IN6_IS_ADDR_6TO4(addr) \
690         (((addr)->_S6_un._S6_u32[0] & 0xffff0000) == 0x20020000)
691 #else /* _BIG_ENDIAN */
692 #define IN6_IS_ADDR_6TO4(addr) \
693         (((addr)->_S6_un._S6_u32[0] & 0x0000ffff) == 0x00000220)
694 #endif /* _BIG_ENDIAN */

696 #define IN6_ARE_6TO4_PREFIX_EQUAL(addr1, addr2) \
697         (((addr1)->_S6_un._S6_u32[0] == (addr2)->_S6_un._S6_u32[0]) && \
698         ((addr1)->_S6_un._S6_u8[4] == (addr2)->_S6_un._S6_u8[4]) && \
699         ((addr1)->_S6_un._S6_u8[5] == (addr2)->_S6_un._S6_u8[5]))

701 /*
702  * IN6_IS_ADDR_LINKSCOPE
703  * Identifies an address as being either link-local, link-local multicast or
704  * node-local multicast.  All types of addresses are considered to be unique
705  * within the scope of a given link.
706  */
707 #define IN6_IS_ADDR_LINKSCOPE(addr) \
708         (IN6_IS_ADDR_LINKLOCAL(addr) || IN6_IS_ADDR_MC_LINKLOCAL(addr) || \
709         IN6_IS_ADDR_MC_NODELOCAL(addr))

711 /*
712  * Useful utility macros for operations with IPv6 addresses
713  * Note: These macros are NOT defined in the RFC2553 or any other
714  * standard specification and are not standard macros that portable
715  * applications should use.
```



```
716  */

718 /*
719  * IN6_V4MAPPED_TO_INADDR
720  * IN6_V4MAPPED_TO_IPADDR
721  *      Assign a IPv4-Mapped IPv6 address to an IPv4 address.
722  *      Note: These macros are NOT defined in RFC2553 or any other standard
723  *      specification and are not macros that portable applications should
724  *      use.
725  *
726  * void IN6_V4MAPPED_TO_INADDR(const in6_addr_t *v6, struct in_addr *v4);
727  * void IN6_V4MAPPED_TO_IPADDR(const in6_addr_t *v6, ipaddr_t v4);
728  *
729  */
730 #define IN6_V4MAPPED_TO_INADDR(v6, v4) \
731         ((v4)->s_addr = (v6)->_S6_un._S6_u32[3])
732 #define IN6_V4MAPPED_TO_IPADDR(v6, v4) \
733         ((v4) = (v6)->_S6_un._S6_u32[3])

735 /*
736  * IN6_INADDR_TO_V4MAPPED
737  * IN6_IPADDR_TO_V4MAPPED
738  *      Assign a IPv4 address address to an IPv6 address as a IPv4-mapped
739  *      address.
740  *      Note: These macros are NOT defined in RFC2553 or any other standard
741  *      specification and are not macros that portable applications should
742  *      use.
743  *
744  * void IN6_INADDR_TO_V4MAPPED(const struct in_addr *v4, in6_addr_t *v6);
745  * void IN6_IPADDR_TO_V4MAPPED(const ipaddr_t v4, in6_addr_t *v6);
746  *
747  */
748 #ifdef _BIG_ENDIAN
749 #define IN6_INADDR_TO_V4MAPPED(v4, v6) \
750         ((v6)->_S6_un._S6_u32[3] = (v4)->s_addr, \
751         (v6)->_S6_un._S6_u32[2] = 0x0000ffff, \
752         (v6)->_S6_un._S6_u32[1] = 0, \
753         (v6)->_S6_un._S6_u32[0] = 0)
754 #define IN6_IPADDR_TO_V4MAPPED(v4, v6) \
755         ((v6)->_S6_un._S6_u32[3] = (v4), \
756         (v6)->_S6_un._S6_u32[2] = 0x0000ffff, \
757         (v6)->_S6_un._S6_u32[1] = 0, \
758         (v6)->_S6_un._S6_u32[0] = 0)
759 #else /* _BIG_ENDIAN */
760 #define IN6_INADDR_TO_V4MAPPED(v4, v6) \
761         ((v6)->_S6_un._S6_u32[3] = (v4)->s_addr, \
762         (v6)->_S6_un._S6_u32[2] = 0xffff0000U, \
763         (v6)->_S6_un._S6_u32[1] = 0, \
764         (v6)->_S6_un._S6_u32[0] = 0)
765 #define IN6_IPADDR_TO_V4MAPPED(v4, v6) \
766         ((v6)->_S6_un._S6_u32[3] = (v4), \
767         (v6)->_S6_un._S6_u32[2] = 0xffff0000U, \
768         (v6)->_S6_un._S6_u32[1] = 0, \
769         (v6)->_S6_un._S6_u32[0] = 0)
770 #endif /* _BIG_ENDIAN */

772 /*
773  * IN6_6TO4_TO_V4ADDR
774  *      Extract the embedded IPv4 address from the prefix to a 6to4 IPv6
775  *      address.
776  *      Note: This macro is NOT defined in RFC2553 or any other standard
777  *      specification and is not a macro that portable applications should
778  *      use.
779  *      Note: we don't use the IPADDR form of the macro because we need
780  *      to do a bytewise copy; the V4ADDR in the 6to4 address is not
781  *      32-bit aligned.
```

```
 *
 * void IN6_6TO4_TO_V4ADDR(const in6_addr_t *v6, struct in_addr *v4);
 *
 */
#define IN6_6TO4_TO_V4ADDR(v6, v4) \
        ((v4)->_S_un._S_un_b.s_b1 = (v6)->_S6_un._S6_u8[2], \
        (v4)->_S_un._S_un_b.s_b2 = (v6)->_S6_un._S6_u8[3],  \
        (v4)->_S_un._S_un_b.s_b3 = (v6)->_S6_un._S6_u8[4],  \
        (v4)->_S_un._S_un_b.s_b4 = (v6)->_S6_un._S6_u8[5])

/*
 * IN6_V4ADDR_TO_6TO4
 *      Given an IPv4 address and an IPv6 address for output, a 6to4 address
 *      will be created from the IPv4 Address.
 *      Note:  This method for creating 6to4 addresses is not standardized
 *      outside of Solaris.  The newly created 6to4 address will be of the form
 *      2002:<V4ADDR>:<SUBNETID>::<HOSTID>, where SUBNETID will equal 0 and
 *      HOSTID will equal 1.
 *
 * void IN6_V4ADDR_TO_6TO4(const struct in_addr *v4, in6_addr_t *v6)
 *
 */
#ifdef _BIG_ENDIAN
#define IN6_V4ADDR_TO_6TO4(v4, v6) \
        ((v6)->_S6_un._S6_u8[0] = 0x20, \
        (v6)->_S6_un._S6_u8[1] = 0x02, \
        (v6)->_S6_un._S6_u8[2] = (v4)->_S_un._S_un_b.s_b1, \
        (v6)->_S6_un._S6_u8[3] = (v4)->_S_un._S_un_b.s_b2, \
        (v6)->_S6_un._S6_u8[4] = (v4)->_S_un._S_un_b.s_b3, \
        (v6)->_S6_un._S6_u8[5] = (v4)->_S_un._S_un_b.s_b4, \
        (v6)->_S6_un._S6_u8[6] = 0, \
        (v6)->_S6_un._S6_u8[7] = 0, \
        (v6)->_S6_un._S6_u32[2] = 0, \
        (v6)->_S6_un._S6_u32[3] = 0x00000001U)
#else
#define IN6_V4ADDR_TO_6TO4(v4, v6) \
        ((v6)->_S6_un._S6_u8[0] = 0x20, \
        (v6)->_S6_un._S6_u8[1] = 0x02, \
        (v6)->_S6_un._S6_u8[2] = (v4)->_S_un._S_un_b.s_b1, \
        (v6)->_S6_un._S6_u8[3] = (v4)->_S_un._S_un_b.s_b2, \
        (v6)->_S6_un._S6_u8[4] = (v4)->_S_un._S_un_b.s_b3, \
        (v6)->_S6_un._S6_u8[5] = (v4)->_S_un._S_un_b.s_b4, \
        (v6)->_S6_un._S6_u8[6] = 0, \
        (v6)->_S6_un._S6_u8[7] = 0, \
        (v6)->_S6_un._S6_u32[2] = 0, \
        (v6)->_S6_un._S6_u32[3] = 0x01000000U)
#endif /* _BIG_ENDIAN */

/*
 * IN6_ARE_ADDR_EQUAL (defined in RFC2292)
 *       Compares if IPv6 addresses are equal.
 * Note: Compares in order of high likelyhood of a miss so we minimize
 * compares. (Current heuristic order, compare in reverse order of
 * uint32_t units)
 *
 * bool  IN6_ARE_ADDR_EQUAL(const struct in6_addr *,
 *                          const struct in6_addr *);
 */
#define IN6_ARE_ADDR_EQUAL(addr1, addr2) \
        (((addr1)->_S6_un._S6_u32[3] == (addr2)->_S6_un._S6_u32[3]) && \
        ((addr1)->_S6_un._S6_u32[2] == (addr2)->_S6_un._S6_u32[2]) && \
        ((addr1)->_S6_un._S6_u32[1] == (addr2)->_S6_un._S6_u32[1]) && \
        ((addr1)->_S6_un._S6_u32[0] == (addr2)->_S6_un._S6_u32[0]))

/*
 * IN6_ARE_PREFIXEDADDR_EQUAL (not defined in RFCs)
```



```
 *      Compares if prefixed parts of IPv6 addresses are equal.
 *
 * uint32_t IN6_MASK_FROM_PREFIX(int, int);
 * bool     IN6_ARE_PREFIXEDADDR_EQUAL(const struct in6_addr *,
 *                                     const struct in6_addr *,
 *                                     int);
 */
#define IN6_MASK_FROM_PREFIX(qoctet, prefix) \
        ((((qoctet) + 1) * 32 < (prefix)) ? 0xFFFFFFFFu : \
        ((((qoctet) * 32) >= (prefix)) ? 0x00000000u : \
        0xFFFFFFFFu << (((qoctet) + 1) * 32 - (prefix))))

#define IN6_ARE_PREFIXEDADDR_EQUAL(addr1, addr2, prefix) \
        (((ntohl((addr1)->_S6_un._S6_u32[0]) & \
        IN6_MASK_FROM_PREFIX(0, prefix)) == \
        (ntohl((addr2)->_S6_un._S6_u32[0]) & \
        IN6_MASK_FROM_PREFIX(0, prefix))) && \
        ((ntohl((addr1)->_S6_un._S6_u32[1]) & \
        IN6_MASK_FROM_PREFIX(1, prefix)) == \
        (ntohl((addr2)->_S6_un._S6_u32[1]) & \
        IN6_MASK_FROM_PREFIX(1, prefix))) && \
        ((ntohl((addr1)->_S6_un._S6_u32[2]) & \
        IN6_MASK_FROM_PREFIX(2, prefix)) == \
        (ntohl((addr2)->_S6_un._S6_u32[2]) & \
        IN6_MASK_FROM_PREFIX(2, prefix))) && \
        ((ntohl((addr1)->_S6_un._S6_u32[3]) & \
        IN6_MASK_FROM_PREFIX(3, prefix)) == \
        (ntohl((addr2)->_S6_un._S6_u32[3]) & \
        IN6_MASK_FROM_PREFIX(3, prefix))))

#endif /* !defined(_XPG4_2) || defined(_XPG6) || defined(__EXTENSIONS__) */


/*
 * Options for use with [gs]etsockopt at the IP level.
 *
 * Note: Some of the IP_ namespace has conflict with and
 * and is exposed through <xti.h>. (It also requires exposing
 * options not implemented). The options with potential
 * for conflicts use #ifndef guards.
 */
#ifndef IP_OPTIONS
#define IP_OPTIONS      1       /* set/get IP per-packet options   */
#endif

#define IP_HDRINCL      2       /* int; header is included with data (raw) */

#ifndef IP_TOS
#define IP_TOS          3       /* int; IP type of service and precedence */
#endif

#ifndef IP_TTL
#define IP_TTL          4       /* int; IP time to live */
#endif

#define IP_RECVOPTS     0x5     /* int; receive all IP options w/datagram */
#define IP_RECVRETOPTS  0x6     /* int; receive IP options for response */
#define IP_RECVDSTADDR  0x7     /* int; receive IP dst addr w/datagram */
#define IP_RETOPTS      0x8     /* ip_opts; set/get IP per-packet options */
#define IP_RECVIF       0x9     /* int; receive the inbound interface index */
#define IP_RECVSLLA     0xa     /* sockaddr_dl; get source link layer address */
#define IP_RECVTTL      0xb     /* uint8_t; get TTL for inbound packet */

#define IP_MULTICAST_IF         0x10    /* set/get IP multicast interface  */
#define IP_MULTICAST_TTL        0x11    /* set/get IP multicast timetolive */
#define IP_MULTICAST_LOOP       0x12    /* set/get IP multicast loopback   */
```

```
914 #define IP_ADD_MEMBERSHIP       0x13    /* add  an IP group membership     */
915 #define IP_DROP_MEMBERSHIP      0x14    /* drop an IP group membership     */
916 #define IP_BLOCK_SOURCE         0x15    /* block   mcast pkts from source  */
917 #define IP_UNBLOCK_SOURCE       0x16    /* unblock mcast pkts from source  */
918 #define IP_ADD_SOURCE_MEMBERSHIP  0x17  /* add  mcast group/source pair    */
919 #define IP_DROP_SOURCE_MEMBERSHIP 0x18  /* drop mcast group/source pair    */
920 #define IP_NEXTHOP              0x19    /* send directly to next hop       */
921 /*
922  * IP_PKTINFO and IP_RECVPKTINFO have same value. Size of argument passed in
923  * is used to differentiate b/w the two.
924  */
925 #define IP_PKTINFO              0x1a    /* specify src address and/or index */
926 #define IP_RECVPKTINFO          0x1a    /* recv dest/matched addr and index */
927 #define IP_DONTFRAG             0x1b    /* don't fragment packets */

929 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
930 /*
931  * Different preferences that can be requested from IPSEC protocols.
932  */
933 #define IP_SEC_OPT              0x22    /* Used to set IPSEC options */
934 #define IPSEC_PREF_NEVER        0x01
935 #define IPSEC_PREF_REQUIRED     0x02
936 #define IPSEC_PREF_UNIQUE       0x04
937 /*
938  * This can be used with the setsockopt() call to set per socket security
939  * options. When the application uses per-socket API, we will reflect
940  * the request on both outbound and inbound packets.
941  */

943 typedef struct ipsec_req {
944         uint_t          ipsr_ah_req;            /* AH request */
945         uint_t          ipsr_esp_req;           /* ESP request */
946         uint_t          ipsr_self_encap_req;    /* Self-Encap request */
947         uint8_t         ipsr_auth_alg;          /* Auth algs for AH */
948         uint8_t         ipsr_esp_alg;           /* Encr algs for ESP */
949         uint8_t         ipsr_esp_auth_alg;      /* Auth algs for ESP */
950 } ipsec_req_t;

952 /*
953  * MCAST_* options are protocol-independent.  The actual definitions
954  * are with the v6 options below; this comment is here to note the
955  * namespace usage.
956  *
957  * #define      MCAST_JOIN_GROUP        0x29
958  * #define      MCAST_LEAVE_GROUP       0x2a
959  * #define      MCAST_BLOCK_SOURCE      0x2b
960  * #define      MCAST_UNBLOCK_SOURCE    0x2c
961  * #define      MCAST_JOIN_SOURCE_GROUP 0x2d
962  * #define      MCAST_LEAVE_SOURCE_GROUP 0x2e
963  */
964 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

966 /*
967  * SunOS private (potentially not portable) IP_ option names
968  */
969 #define IP_BOUND_IF             0x41    /* bind socket to an ifindex       */
970 #define IP_UNSPEC_SRC           0x42    /* use unspecified source address  */
971 #define IP_BROADCAST_TTL        0x43    /* use specific TTL for broadcast  */
972 /* can be reused                0x44 */
973 #define IP_DHCPINIT_IF          0x45    /* accept all unicast DHCP traffic */

975 /*
976  * Option values and names (when !_XPG5) shared with <xti_inet.h>
977  */
978 #ifndef IP_REUSEADDR
979 #define IP_REUSEADDR            0x104
```



```
980 #endif

982 #ifndef IP_DONTROUTE
983 #define IP_DONTROUTE            0x105
984 #endif

986 #ifndef IP_BROADCAST
987 #define IP_BROADCAST            0x106
988 #endif

990 /*
991  * The following option values are reserved by <xti_inet.h>
992  *
993  * T_IP_OPTIONS 0x107    -  IP per-packet options
994  * T_IP_TOS     0x108    -  IP per packet type of service
995  */

997 /*
998  * Default value constants for multicast attributes controlled by
999  * IP*_MULTICAST_LOOP and IP*_MULTICAST_{TTL,HOPS} options.
1000  */
1001 #define IP_DEFAULT_MULTICAST_TTL  1     /* normally limit m'casts to 1 hop */
1002 #define IP_DEFAULT_MULTICAST_LOOP 1     /* normally hear sends if a member */

1004 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
1005 /*
1006  * Argument structure for IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP.
1007  */
1008 struct ip_mreq {
1009         struct in_addr  imr_multiaddr;  /* IP multicast address of group */
1010         struct in_addr  imr_interface;  /* local IP address of interface */
1011 };

1013 /*
1014  * Argument structure for IP_BLOCK_SOURCE, IP_UNBLOCK_SOURCE,
1015  * IP_ADD_SOURCE_MEMBERSHIP, and IP_DROP_SOURCE_MEMBERSHIP.
1016  */
1017 struct ip_mreq_source {
1018         struct in_addr  imr_multiaddr;  /* IP address of group */
1019         struct in_addr  imr_sourceaddr; /* IP address of source */
1020         struct in_addr  imr_interface;  /* IP address of interface */
1021 };

1023 /*
1024  * Argument structure for IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP on
1025  * IPv6 addresses.
1026  */
1027 struct ipv6_mreq {
1028         struct in6_addr ipv6mr_multiaddr;       /* IPv6 multicast addr */
1029         unsigned int    ipv6mr_interface;       /* interface index */
1030 };

1032 /*
1033  * Use #pragma pack() construct to force 32-bit alignment on amd64.
1034  * This is needed to keep the structure size and offsets consistent
1035  * between a 32-bit app and the 64-bit amd64 kernel in structures
1036  * where 64-bit alignment would create gaps (in this case, structures
1037  * which have a uint32_t followed by a struct sockaddr_storage).
1038  */
1039 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1040 #pragma pack(4)
1041 #endif

1043 /*
1044  * Argument structure for MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP.
1045  */
```

```
1046 struct group_req {
1047         uint32_t                gr_interface;   /* interface index */
1048         struct sockaddr_storage gr_group;       /* group address */
1049 };

1051 /*
1052  * Argument structure for MCAST_BLOCK_SOURCE, MCAST_UNBLOCK_SOURCE,
1053  * MCAST_JOIN_SOURCE_GROUP, MCAST_LEAVE_SOURCE_GROUP.
1054  */
1055 struct group_source_req {
1056         uint32_t                gsr_interface;  /* interface index */
1057         struct sockaddr_storage gsr_group;      /* group address */
1058         struct sockaddr_storage gsr_source;     /* source address */
1059 };

1061 /*
1062  * Argument for SIOC[GS]MSFILTER ioctls
1063  */
1064 struct group_filter {
1065         uint32_t                gf_interface;   /* interface index */
1066         struct sockaddr_storage gf_group;       /* multicast address */
1067         uint32_t                gf_fmode;       /* filter mode */
1068         uint32_t                gf_numsrc;      /* number of sources */
1069         struct sockaddr_storage gf_slist[1];    /* source address */
1070 };

1072 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1073 #pragma pack()
1074 #endif

1076 #define GROUP_FILTER_SIZE(numsrc) \
1077         (sizeof (struct group_filter) - sizeof (struct sockaddr_storage) \
1078         + (numsrc) * sizeof (struct sockaddr_storage))

1080 /*
1081  * Argument for SIOC[GS]IPMSFILTER ioctls (IPv4-specific)
1082  */
1083 struct ip_msfilter {
1084         struct in_addr  imsf_multiaddr; /* IP multicast address of group */
1085         struct in_addr  imsf_interface; /* local IP address of interface */
1086         uint32_t        imsf_fmode;     /* filter mode */
1087         uint32_t        imsf_numsrc;    /* number of sources in src_list */
1088         struct in_addr  imsf_slist[1];  /* start of source list */
1089 };

1091 #define IP_MSFILTER_SIZE(numsrc) \
1092         (sizeof (struct ip_msfilter) - sizeof (struct in_addr) \
1093         + (numsrc) * sizeof (struct in_addr))

1095 /*
1096  * Multicast source filter manipulation functions in libsocket;
1097  * defined in RFC 3678.
1098  */
1099 int setsourcefilter(int, uint32_t, struct sockaddr *, socklen_t, uint32_t,
1100                         uint_t, struct sockaddr_storage *);

1102 int getsourcefilter(int, uint32_t, struct sockaddr *, socklen_t, uint32_t *,
1103                         uint_t *, struct sockaddr_storage *);

1105 int setipv4sourcefilter(int, struct in_addr, struct in_addr, uint32_t,
1106                         uint32_t, struct in_addr *);

1108 int getipv4sourcefilter(int, struct in_addr, struct in_addr, uint32_t *,
1109                         uint32_t *, struct in_addr *);

1111 /*
```



```
1112  * Definitions needed for [gs]etsourcefilter(), [gs]etipv4sourcefilter()
1113  */
1114 #define MCAST_INCLUDE   1
1115 #define MCAST_EXCLUDE   2

1117 /*
1118  * Argument struct for IP_PKTINFO option
1119  */
1120 typedef struct in_pktinfo {
1121         unsigned int            ipi_ifindex;    /* send/recv interface index */
1122         struct in_addr          ipi_spec_dst;   /* matched source address */
1123         struct in_addr          ipi_addr;       /* src/dst address in IP hdr */
1124 } in_pktinfo_t;

1126 /*
1127  * Argument struct for IPV6_PKTINFO option
1128  */
1129 struct in6_pktinfo {
1130         struct in6_addr         ipi6_addr;      /* src/dst IPv6 address */
1131         unsigned int            ipi6_ifindex;   /* send/recv interface index */
1132 };

1134 /*
1135  * Argument struct for IPV6_MTUINFO option
1136  */
1137 struct ip6_mtuinfo {
1138         struct sockaddr_in6     ip6m_addr; /* dst address including zone ID */
1139         uint32_t                ip6m_mtu;  /* path MTU in host byte order */
1140 };

1142 /*
1143  * IPv6 routing header types
1144  */
1145 #define IPV6_RTHDR_TYPE_0       0

1147 extern socklen_t inet6_rth_space(int type, int segments);
1148 extern void *inet6_rth_init(void *bp, socklen_t bp_len, int type, int segments);
1149 extern int inet6_rth_add(void *bp, const struct in6_addr *addr);
1150 extern int inet6_rth_reverse(const void *in, void *out);
1151 extern int inet6_rth_segments(const void *bp);
1152 extern struct in6_addr *inet6_rth_getaddr(const void *bp, int index);

1154 extern int inet6_opt_init(void *extbuf, socklen_t extlen);
1155 extern int inet6_opt_append(void *extbuf, socklen_t extlen, int offset,
1156         uint8_t type, socklen_t len, uint_t align, void **databufp);
1157 extern int inet6_opt_finish(void *extbuf, socklen_t extlen, int offset);
1158 extern int inet6_opt_set_val(void *databuf, int offset, void *val,
1159         socklen_t vallen);
1160 extern int inet6_opt_next(void *extbuf, socklen_t extlen, int offset,
1161         uint8_t *typep, socklen_t *lenp, void **databufp);
1162 extern int inet6_opt_find(void *extbufp, socklen_t extlen, int offset,
1163         uint8_t type, socklen_t *lenp, void **databufp);
1164 extern int inet6_opt_get_val(void *databuf, int offset, void *val,
1165         socklen_t vallen);
1166 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

1168 /*
1169  * Argument structure for IP_ADD_PROXY_ADDR.
1170  * Note that this is an unstable, experimental interface. It may change
1171  * later. Don't use it unless you know what it is.
1172  */
1173 typedef struct {
1174         struct in_addr  in_prefix_addr;
1175         unsigned int    in_prefix_len;
1176 } in_prefix_t;
```

```
1179 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
1180 /*
1181  * IPv6 options
1182  */
1183 #define IPV6_UNICAST_HOPS       0x5     /* hop limit value for unicast */
1184                                         /* packets. */
1185                                         /* argument type: uint_t */
1186 #define IPV6_MULTICAST_IF       0x6     /* outgoing interface for */
1187                                         /* multicast packets. */
1188                                         /* argument type: struct in6_addr */
1189 #define IPV6_MULTICAST_HOPS     0x7     /* hop limit value to use for */
1190                                         /* multicast packets. */
1191                                         /* argument type: uint_t */
1192 #define IPV6_MULTICAST_LOOP     0x8     /* enable/disable delivery of */
1193                                         /* multicast packets on same socket. */
1194                                         /* argument type: uint_t */
1195 #define IPV6_JOIN_GROUP         0x9     /* join an IPv6 multicast group. */
1196                                         /* argument type: struct ipv6_mreq */
1197 #define IPV6_LEAVE_GROUP        0xa     /* leave an IPv6 multicast group */
1198                                         /* argument type: struct ipv6_mreq */
1199 /*
1200  * IPV6_ADD_MEMBERSHIP and IPV6_DROP_MEMBERSHIP are being kept
1201  * for backward compatibility. They have the same meaning as IPV6_JOIN_GROUP
1202  * and IPV6_LEAVE_GROUP respectively.
1203  */
1204 #define IPV6_ADD_MEMBERSHIP     0x9     /* join an IPv6 multicast group. */
1205                                         /* argument type: struct ipv6_mreq */
1206 #define IPV6_DROP_MEMBERSHIP    0xa     /* leave an IPv6 multicast group */
1207                                         /* argument type: struct ipv6_mreq */

1209 #define IPV6_PKTINFO            0xb     /* addr plus interface index */
1210                                         /* arg type: "struct in6_pktingo" - */
1211 #define IPV6_HOPLIMIT           0xc     /* hoplimit for datagram */
1212 #define IPV6_NEXTHOP            0xd     /* next hop address  */
1213 #define IPV6_HOPOPTS            0xe     /* hop by hop options */
1214 #define IPV6_DSTOPTS            0xf     /* destination options - after */
1215                                         /* the routing header */
1216 #define IPV6_RTHDR              0x10    /* routing header  */
1217 #define IPV6_RTHDRDSTOPTS       0x11    /* destination options - before */
1218                                         /* the routing header */
1219 #define IPV6_RECVPKTINFO        0x12    /* enable/disable IPV6_PKTINFO */
1220 #define IPV6_RECVHOPLIMIT       0x13    /* enable/disable IPV6_HOPLIMIT */
1221 #define IPV6_RECVHOPOPTS        0x14    /* enable/disable IPV6_HOPOPTS */

1223 /*
1224  * This options exists for backwards compatability and should no longer be
1225  * used.  Use IPV6_RECVDSTOPTS instead.
1226  */
1227 #define _OLD_IPV6_RECVDSTOPTS   0x15

1229 #define IPV6_RECVRTHDR          0x16    /* enable/disable IPV6_RTHDR */

1231 /*
1232  * enable/disable IPV6_RTHDRDSTOPTS.  Now obsolete.  IPV6_RECVDSTOPTS enables
1233  * the receipt of both headers.
1234  */
1235 #define IPV6_RECVRTHDRDSTOPTS   0x17

1237 #define IPV6_CHECKSUM           0x18    /* Control checksum on raw sockets */
1238 #define IPV6_RECVTCLASS         0x19    /* enable/disable IPV6_CLASS */
1239 #define IPV6_USE_MIN_MTU        0x20    /* send packets with minimum MTU */
1240 #define IPV6_DONTFRAG           0x21    /* don't fragment packets */
1241 #define IPV6_SEC_OPT            0x22    /* Used to set IPSEC options */
1242 #define IPV6_SRC_PREFERENCES    0x23    /* Control socket's src addr select */
1243 #define IPV6_RECVPATHMTU        0x24    /* receive PMTU info */
```



```
1244 #define IPV6_PATHMTU            0x25    /* get the PMTU */
1245 #define IPV6_TCLASS             0x26    /* traffic class */
1246 #define IPV6_V6ONLY             0x27    /* v6 only socket option */

1248 /*
1249  * enable/disable receipt of both both IPV6_DSTOPTS headers.
1250  */
1251 #define IPV6_RECVDSTOPTS        0x28

1253 /*
1254  * protocol-independent multicast membership options.
1255  */
1256 #define MCAST_JOIN_GROUP        0x29    /* join group for all sources */
1257 #define MCAST_LEAVE_GROUP       0x2a    /* leave group */
1258 #define MCAST_BLOCK_SOURCE      0x2b    /* block specified source */
1259 #define MCAST_UNBLOCK_SOURCE    0x2c    /* unblock specified source */
1260 #define MCAST_JOIN_SOURCE_GROUP 0x2d    /* join group for specified source */
1261 #define MCAST_LEAVE_SOURCE_GROUP 0x2e   /* leave source/group pair */

1263 /* 32Bit field for IPV6_SRC_PREFERENCES */
1264 #define IPV6_PREFER_SRC_HOME            0x00000001
1265 #define IPV6_PREFER_SRC_COA             0x00000002
1266 #define IPV6_PREFER_SRC_PUBLIC          0x00000004
1267 #define IPV6_PREFER_SRC_TMP             0x00000008
1268 #define IPV6_PREFER_SRC_NONCGA          0x00000010
1269 #define IPV6_PREFER_SRC_CGA             0x00000020

1271 #define IPV6_PREFER_SRC_MIPMASK (IPV6_PREFER_SRC_HOME | IPV6_PREFER_SRC_COA)
1272 #define IPV6_PREFER_SRC_MIPDEFAULT      IPV6_PREFER_SRC_HOME
1273 #define IPV6_PREFER_SRC_TMPMASK (IPV6_PREFER_SRC_PUBLIC | IPV6_PREFER_SRC_TMP)
1274 #define IPV6_PREFER_SRC_TMPDEFAULT      IPV6_PREFER_SRC_PUBLIC
1275 #define IPV6_PREFER_SRC_CGAMASK (IPV6_PREFER_SRC_NONCGA | IPV6_PREFER_SRC_CGA)
1276 #define IPV6_PREFER_SRC_CGADEFAULT      IPV6_PREFER_SRC_NONCGA

1278 #define IPV6_PREFER_SRC_MASK (IPV6_PREFER_SRC_MIPMASK |\
1279         IPV6_PREFER_SRC_TMPMASK | IPV6_PREFER_SRC_CGAMASK)

1281 #define IPV6_PREFER_SRC_DEFAULT (IPV6_PREFER_SRC_MIPDEFAULT |\
1282         IPV6_PREFER_SRC_TMPDEFAULT | IPV6_PREFER_SRC_CGADEFAULT)

1284 /*
1285  * SunOS private (potentially not portable) IPV6_ option names
1286  */
1287 #define IPV6_BOUND_IF           0x41    /* bind to an ifindex */
1288 #define IPV6_UNSPEC_SRC         0x42    /* source of packets set to */
1289                                         /* unspecified (all zeros) */

1291 /*
1292  * Miscellaneous IPv6 constants.
1293  */
1294 #define INET_ADDRSTRLEN         16      /* max len IPv4 addr in ascii dotted */
1295                                         /* decimal notation. */
1296 #define INET6_ADDRSTRLEN        46      /* max len of IPv6 addr in ascii */
1297                                         /* standard colon-hex notation. */
1298 #define IPV6_PAD1_OPT           0       /* pad byte in IPv6 extension hdrs */

1300 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

1302 /*
1303  * Extern declarations for pre-defined global const variables
1304  */
1305 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
1306 #ifndef _KERNEL
1307 #ifdef __STDC__
1308 extern const struct in6_addr in6addr_any;
1309 extern const struct in6_addr in6addr_loopback;
```

```
#else
extern struct in6_addr in6addr_any;
extern struct in6_addr in6addr_loopback;
#endif
#endif
#endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

#ifdef  __cplusplus
}
#endif

#endif  /* _NETINET_IN_H */
```

```
**********************************************************
    9019 Mon Jul  9 14:38:22 2012
new/usr/src/uts/common/sys/netstack.h
dccp: starting module template
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
#ifndef _SYS_NETSTACK_H
#define _SYS_NETSTACK_H

#include <sys/kstat.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * This allows various pieces in and around IP to have a separate instance
 * for each instance of IP. This is used to support zones that have an
 * exclusive stack.
 * Pieces of software far removed from IP (e.g., kernel software
 * sitting on top of TCP or UDP) probably should not use the netstack
 * support; if such software wants to support separate zones it
 * can do that using the zones framework (zone_key_create() etc)
 * whether there is a shared IP stack or and exclusive IP stack underneath.
 */

/*
 * Each netstack has an identifier. We reuse the zoneid allocation for
 * this but have a separate typedef. Thus the shared stack (used by
 * the global zone and other shared stack zones) have a zero ID, and
 * the exclusive stacks have a netstackid that is the same as their zoneid.
 */
typedef id_t    netstackid_t;

#define GLOBAL_NETSTACKID       0

/*
 * One for each module which uses netstack support.
 * Used in netstack_register().
 *
 * The order of these is important for some modules both for
 * the creation (which done in ascending order) and destruction (which is
```



```
 * done in in decending order).
 */
#define NS_ALL          -1      /* Match all */
#define NS_DLS          0
#define NS_IPTUN        1
#define NS_STR          2       /* autopush list etc */
#define NS_HOOK         3
#define NS_NETI         4
#define NS_ARP          5
#define NS_IP           6
#define NS_ICMP         7
#define NS_UDP          8
#define NS_TCP          9
#define NS_SCTP         10
#define NS_RTS          11
#define NS_IPSEC        12
#define NS_KEYSOCK      13
#define NS_SPDSOCK      14
#define NS_IPSECAH      15
#define NS_IPSECESP     16
#define NS_IPNET        17
#define NS_ILB          18
#define NS_DCCP         19
#define NS_MAX          (NS_DCCP+1)
#define NS_MAX          (NS_ILB+1)

/*
 * State maintained for each module which tracks the state of
 * the create, shutdown and destroy callbacks.
 *
 * Keeps track of pending actions to avoid holding locks when
 * calling into the create/shutdown/destroy functions in the module.
 */
#ifdef _KERNEL
typedef struct {
        uint16_t        nms_flags;
        kcondvar_t      nms_cv;
} nm_state_t;

/*
 * nms_flags
 */
#define NSS_CREATE_NEEDED       0x0001
#define NSS_CREATE_INPROGRESS   0x0002
#define NSS_CREATE_COMPLETED    0x0004
#define NSS_SHUTDOWN_NEEDED     0x0010
#define NSS_SHUTDOWN_INPROGRESS 0x0020
#define NSS_SHUTDOWN_COMPLETED  0x0040
#define NSS_DESTROY_NEEDED      0x0100
#define NSS_DESTROY_INPROGRESS  0x0200
#define NSS_DESTROY_COMPLETED   0x0400

#define NSS_CREATE_ALL  \
        (NSS_CREATE_NEEDED|NSS_CREATE_INPROGRESS|NSS_CREATE_COMPLETED)
#define NSS_SHUTDOWN_ALL        \
        (NSS_SHUTDOWN_NEEDED|NSS_SHUTDOWN_INPROGRESS|NSS_SHUTDOWN_COMPLETED)
#define NSS_DESTROY_ALL \
        (NSS_DESTROY_NEEDED|NSS_DESTROY_INPROGRESS|NSS_DESTROY_COMPLETED)

#define NSS_ALL_INPROGRESS      \
        (NSS_CREATE_INPROGRESS|NSS_SHUTDOWN_INPROGRESS|NSS_DESTROY_INPROGRESS)
#else
/* User-level compile like IP Filter needs a netstack_t. Dummy */
typedef uint_t nm_state_t;
#endif /* _KERNEL */
```

```
127 /*
128  * One for every netstack in the system.
129  * We use a union so that the compilar and lint can provide type checking -
130  * in principle we could have
131  * #define      netstack_arp            netstack_modules[NS_ARP]
132  * etc, but that would imply void * types hence no type checking by the
133  * compiler.
134  *
135  * All the fields in netstack_t except netstack_next are protected by
136  * netstack_lock. netstack_next is protected by netstack_g_lock.
137  */
138 struct netstack {
139         union {
140                 void    *nu_modules[NS_MAX];
141                 struct {
142                         struct dls_stack        *nu_dls;
143                         struct iptun_stack      *nu_iptun;
144                         struct str_stack        *nu_str;
145                         struct hook_stack       *nu_hook;
146                         struct neti_stack       *nu_neti;
147                         struct arp_stack        *nu_arp;
148                         struct ip_stack         *nu_ip;
149                         struct icmp_stack       *nu_icmp;
150                         struct udp_stack        *nu_udp;
151                         struct tcp_stack        *nu_tcp;
152                         struct sctp_stack       *nu_sctp;
153                         struct rts_stack        *nu_rts;
154                         struct ipsec_stack      *nu_ipsec;
155                         struct keysock_stack    *nu_keysock;
156                         struct spd_stack        *nu_spdsock;
157                         struct ipsecah_stack    *nu_ipsecah;
158                         struct ipsecesp_stack   *nu_ipsecesp;
159                         struct ipnet_stack      *nu_ipnet;
160                         struct ilb_stack        *nu_ilb;
161                         struct dccp_stack       *nu_dccp;
162 #endif /* ! codereview */
163                 } nu_s;
164         } netstack_u;
165 #define netstack_modules        netstack_u.nu_modules
166 #define netstack_dls            netstack_u.nu_s.nu_dls
167 #define netstack_iptun          netstack_u.nu_s.nu_iptun
168 #define netstack_str            netstack_u.nu_s.nu_str
169 #define netstack_hook           netstack_u.nu_s.nu_hook
170 #define netstack_neti           netstack_u.nu_s.nu_neti
171 #define netstack_arp            netstack_u.nu_s.nu_arp
172 #define netstack_ip             netstack_u.nu_s.nu_ip
173 #define netstack_icmp           netstack_u.nu_s.nu_icmp
174 #define netstack_udp            netstack_u.nu_s.nu_udp
175 #define netstack_tcp            netstack_u.nu_s.nu_tcp
176 #define netstack_sctp           netstack_u.nu_s.nu_sctp
177 #define netstack_rts            netstack_u.nu_s.nu_rts
178 #define netstack_ipsec          netstack_u.nu_s.nu_ipsec
179 #define netstack_keysock        netstack_u.nu_s.nu_keysock
180 #define netstack_spdsock        netstack_u.nu_s.nu_spdsock
181 #define netstack_ipsecah        netstack_u.nu_s.nu_ipsecah
182 #define netstack_ipsecesp       netstack_u.nu_s.nu_ipsecesp
183 #define netstack_ipnet          netstack_u.nu_s.nu_ipnet
184 #define netstack_ilb            netstack_u.nu_s.nu_ilb
185 #define netstack_dccp           netstack_u.nu_s.nu_dccp
186 #endif /* ! codereview */

188         nm_state_t      netstack_m_state[NS_MAX]; /* module state */

190         kmutex_t        netstack_lock;
191         struct netstack *netstack_next;
192         netstackid_t    netstack_stackid;
```



```
193         int             netstack_numzones;      /* Number of zones using this */
194         int             netstack_refcnt;        /* Number of hold-rele */
195         int             netstack_flags; /* See below */

197 #ifdef _KERNEL
198         /* Needed to ensure that we run the callback functions in order */
199         kcondvar_t      netstack_cv;
200 #endif
201 };
202 typedef struct netstack netstack_t;

204 /* netstack_flags values */
205 #define NSF_UNINIT              0x01            /* Not initialized */
206 #define NSF_CLOSING             0x02            /* Going away */
207 #define NSF_ZONE_CREATE         0x04            /* create callbacks inprog */
208 #define NSF_ZONE_SHUTDOWN       0x08            /* shutdown callbacks */
209 #define NSF_ZONE_DESTROY        0x10            /* destroy callbacks */

211 #define NSF_ZONE_INPROGRESS     \
212         (NSF_ZONE_CREATE|NSF_ZONE_SHUTDOWN|NSF_ZONE_DESTROY)

214 /*
215  * One for each of the NS_* values.
216  */
217 struct netstack_registry {
218         int             nr_flags;       /* 0 if nothing registered */
219         void            *(*nr_create)(netstackid_t, netstack_t *);
220         void            (*nr_shutdown)(netstackid_t, void *);
221         void            (*nr_destroy)(netstackid_t, void *);
222 };

224 /* nr_flags values */
225 #define NRF_REGISTERED  0x01
226 #define NRF_DYING       0x02    /* No new creates */

228 /*
229  * To support kstat_create_netstack() using kstat_add_zone we need
230  * to track both
231  *  - all zoneids that use the global/shared stack
232  *  - all kstats that have been added for the shared stack
233  */

235 extern void netstack_init(void);
236 extern void netstack_hold(netstack_t *);
237 extern void netstack_rele(netstack_t *);
238 extern netstack_t *netstack_find_by_cred(const cred_t *);
239 extern netstack_t *netstack_find_by_stackid(netstackid_t);
240 extern netstack_t *netstack_find_by_zoneid(zoneid_t);

242 extern zoneid_t netstackid_to_zoneid(netstackid_t);
243 extern zoneid_t netstack_get_zoneid(netstack_t *);
244 extern netstackid_t zoneid_to_netstackid(zoneid_t);

246 extern netstack_t *netstack_get_current(void);

248 /*
249  * Register interest in changes to the set of netstacks.
250  * The createfn and destroyfn are required, but the shutdownfn can be
251  * NULL.
252  * Note that due to the current zsd implementation, when the create
253  * function is called the zone isn't fully present, thus functions
254  * like zone_find_by_* will fail, hence the create function can not
255  * use many zones kernel functions including zcmn_err().
256  */
257 extern void     netstack_register(int,
258     void *(*)(netstackid_t, netstack_t *),
```

```
259     void (*)(netstackid_t, void *),
260     void (*)(netstackid_t, void *));
261 extern void     netstack_unregister(int);
262 extern kstat_t  *kstat_create_netstack(char *, int, char *, char *, uchar_t,
263     uint_t, uchar_t, netstackid_t);
264 extern void     kstat_delete_netstack(kstat_t *, netstackid_t);

266 /*
267  * Simple support for walking all the netstacks.
268  * The caller of netstack_next() needs to call netstack_rele() when
269  * done with a netstack.
270  */
271 typedef int     netstack_handle_t;

273 extern void     netstack_next_init(netstack_handle_t *);
274 extern void     netstack_next_fini(netstack_handle_t *);
275 extern netstack_t       *netstack_next(netstack_handle_t *);

277 #ifdef  __cplusplus
278 }
279 #endif


282 #endif  /* _SYS_NETSTACK_H */
```

```
**********************************************************
   16746 Mon Jul  9 14:38:23 2012
new/usr/src/uts/common/sys/sdt.h
dccp: connect
**********************************************************
______unchanged_portion_omitted_

 141 #define DTRACE_SCHED(name)                                              \
 142         DTRACE_PROBE(__sched_##name);

 144 #define DTRACE_SCHED1(name, type1, arg1)                                \
 145         DTRACE_PROBE1(__sched_##name, type1, arg1);

 147 #define DTRACE_SCHED2(name, type1, arg1, type2, arg2)                   \
 148         DTRACE_PROBE2(__sched_##name, type1, arg1, type2, arg2);

 150 #define DTRACE_SCHED3(name, type1, arg1, type2, arg2, type3, arg3)      \
 151         DTRACE_PROBE3(__sched_##name, type1, arg1, type2, arg2, type3, arg3);

 153 #define DTRACE_SCHED4(name, type1, arg1, type2, arg2,                   \
 154     type3, arg3, type4, arg4)                                           \
 155         DTRACE_PROBE4(__sched_##name, type1, arg1, type2, arg2,         \
 156             type3, arg3, type4, arg4);

 158 #define DTRACE_PROC(name)                                               \
 159         DTRACE_PROBE(__proc_##name);

 161 #define DTRACE_PROC1(name, type1, arg1)                                 \
 162         DTRACE_PROBE1(__proc_##name, type1, arg1);

 164 #define DTRACE_PROC2(name, type1, arg1, type2, arg2)                    \
 165         DTRACE_PROBE2(__proc_##name, type1, arg1, type2, arg2);

 167 #define DTRACE_PROC3(name, type1, arg1, type2, arg2, type3, arg3)       \
 168         DTRACE_PROBE3(__proc_##name, type1, arg1, type2, arg2, type3, arg3);

 170 #define DTRACE_PROC4(name, type1, arg1, type2, arg2,                    \
 171     type3, arg3, type4, arg4)                                           \
 172         DTRACE_PROBE4(__proc_##name, type1, arg1, type2, arg2,          \
 173             type3, arg3, type4, arg4);

 175 #define DTRACE_IO(name)                                                 \
 176         DTRACE_PROBE(__io_##name);

 178 #define DTRACE_IO1(name, type1, arg1)                                   \
 179         DTRACE_PROBE1(__io_##name, type1, arg1);

 181 #define DTRACE_IO2(name, type1, arg1, type2, arg2)                      \
 182         DTRACE_PROBE2(__io_##name, type1, arg1, type2, arg2);

 184 #define DTRACE_IO3(name, type1, arg1, type2, arg2, type3, arg3) \
 185         DTRACE_PROBE3(__io_##name, type1, arg1, type2, arg2, type3, arg3);

 187 #define DTRACE_IO4(name, type1, arg1, type2, arg2,                      \
 188     type3, arg3, type4, arg4)                                           \
 189         DTRACE_PROBE4(__io_##name, type1, arg1, type2, arg2,            \
 190             type3, arg3, type4, arg4);

 192 #define DTRACE_ISCSI_2(name, type1, arg1, type2, arg2)                  \
 193         DTRACE_PROBE2(__iscsi_##name, type1, arg1, type2, arg2);

 195 #define DTRACE_ISCSI_3(name, type1, arg1, type2, arg2, type3, arg3)     \
 196         DTRACE_PROBE3(__iscsi_##name, type1, arg1, type2, arg2, type3, arg3);

 198 #define DTRACE_ISCSI_4(name, type1, arg1, type2, arg2,                  \
 199     type3, arg3, type4, arg4)                                           \
```



```
 200         DTRACE_PROBE4(__iscsi_##name, type1, arg1, type2, arg2,         \
 201             type3, arg3, type4, arg4);

 203 #define DTRACE_ISCSI_5(name, type1, arg1, type2, arg2,                  \
 204     type3, arg3, type4, arg4, type5, arg5)                              \
 205         DTRACE_PROBE5(__iscsi_##name, type1, arg1, type2, arg2,         \
 206             type3, arg3, type4, arg4, type5, arg5);

 208 #define DTRACE_ISCSI_6(name, type1, arg1, type2, arg2,                  \
 209     type3, arg3, type4, arg4, type5, arg5, type6, arg6)                 \
 210         DTRACE_PROBE6(__iscsi_##name, type1, arg1, type2, arg2,         \
 211             type3, arg3, type4, arg4, type5, arg5, type6, arg6);

 213 #define DTRACE_ISCSI_7(name, type1, arg1, type2, arg2,                  \
 214     type3, arg3, type4, arg4, type5, arg5, type6, arg6, type7, arg7)    \
 215         DTRACE_PROBE7(__iscsi_##name, type1, arg1, type2, arg2,         \
 216             type3, arg3, type4, arg4, type5, arg5, type6, arg6,         \
 217             type7, arg7);

 219 #define DTRACE_ISCSI_8(name, type1, arg1, type2, arg2,                  \
 220     type3, arg3, type4, arg4, type5, arg5, type6, arg6,                 \
 221     type7, arg7, type8, arg8)                                           \
 222         DTRACE_PROBE8(__iscsi_##name, type1, arg1, type2, arg2,         \
 223             type3, arg3, type4, arg4, type5, arg5, type6, arg6,         \
 224             type7, arg7, type8, arg8);

 226 #define DTRACE_NFSV3_3(name, type1, arg1, type2, arg2,                  \
 227     type3, arg3)                                                        \
 228         DTRACE_PROBE3(__nfsv3_##name, type1, arg1, type2, arg2,         \
 229             type3, arg3);
 230 #define DTRACE_NFSV3_4(name, type1, arg1, type2, arg2,                  \
 231     type3, arg3, type4, arg4)                                           \
 232         DTRACE_PROBE4(__nfsv3_##name, type1, arg1, type2, arg2,         \
 233             type3, arg3, type4, arg4);

 235 #define DTRACE_NFSV4_1(name, type1, arg1) \
 236         DTRACE_PROBE1(__nfsv4_##name, type1, arg1);

 238 #define DTRACE_NFSV4_2(name, type1, arg1, type2, arg2) \
 239         DTRACE_PROBE2(__nfsv4_##name, type1, arg1, type2, arg2);

 241 #define DTRACE_NFSV4_3(name, type1, arg1, type2, arg2, type3, arg3) \
 242         DTRACE_PROBE3(__nfsv4_##name, type1, arg1, type2, arg2, type3, arg3);

 244 #define DTRACE_SMB_1(name, type1, arg1) \
 245         DTRACE_PROBE1(__smb_##name, type1, arg1);

 247 #define DTRACE_SMB_2(name, type1, arg1, type2, arg2) \
 248         DTRACE_PROBE2(__smb_##name, type1, arg1, type2, arg2);

 250 #define DTRACE_IP(name)                                                 \
 251         DTRACE_PROBE(__ip_##name);

 253 #define DTRACE_IP1(name, type1, arg1)                                   \
 254         DTRACE_PROBE1(__ip_##name, type1, arg1);

 256 #define DTRACE_IP2(name, type1, arg1, type2, arg2)                      \
 257         DTRACE_PROBE2(__ip_##name, type1, arg1, type2, arg2);

 259 #define DTRACE_IP3(name, type1, arg1, type2, arg2, type3, arg3) \
 260         DTRACE_PROBE3(__ip_##name, type1, arg1, type2, arg2, type3, arg3);

 262 #define DTRACE_IP4(name, type1, arg1, type2, arg2,                      \
 263     type3, arg3, type4, arg4)                                           \
 264         DTRACE_PROBE4(__ip_##name, type1, arg1, type2, arg2,            \
 265             type3, arg3, type4, arg4);
```

```
#define DTRACE_IP5(name, type1, arg1, type2, arg2,                      \
    type3, arg3, type4, arg4, type5, arg5)                              \
        DTRACE_PROBE5(__ip_##name, type1, arg1, type2, arg2,            \
            type3, arg3, type4, arg4, type5, arg5);

#define DTRACE_IP6(name, type1, arg1, type2, arg2,                      \
    type3, arg3, type4, arg4, type5, arg5, type6, arg6)                 \
        DTRACE_PROBE6(__ip_##name, type1, arg1, type2, arg2,            \
            type3, arg3, type4, arg4, type5, arg5, type6, arg6);

#define DTRACE_IP7(name, type1, arg1, type2, arg2, type3, arg3,         \
    type4, arg4, type5, arg5, type6, arg6, type7, arg7)                 \
        DTRACE_PROBE7(__ip_##name, type1, arg1, type2, arg2,            \
            type3, arg3, type4, arg4, type5, arg5, type6, arg6,         \
            type7, arg7);

#define DTRACE_TCP(name)                                                \
        DTRACE_PROBE(__tcp_##name);

#define DTRACE_TCP1(name, type1, arg1)                                  \
        DTRACE_PROBE1(__tcp_##name, type1, arg1);

#define DTRACE_TCP2(name, type1, arg1, type2, arg2)                     \
        DTRACE_PROBE2(__tcp_##name, type1, arg1, type2, arg2);

#define DTRACE_TCP3(name, type1, arg1, type2, arg2, type3, arg3)        \
        DTRACE_PROBE3(__tcp_##name, type1, arg1, type2, arg2, type3, arg3);

#define DTRACE_TCP4(name, type1, arg1, type2, arg2,                     \
    type3, arg3, type4, arg4)                                           \
        DTRACE_PROBE4(__tcp_##name, type1, arg1, type2, arg2,           \
            type3, arg3, type4, arg4);

#define DTRACE_TCP5(name, type1, arg1, type2, arg2,                     \
    type3, arg3, type4, arg4, type5, arg5)                              \
        DTRACE_PROBE5(__tcp_##name, type1, arg1, type2, arg2,           \
            type3, arg3, type4, arg4, type5, arg5);

#define DTRACE_TCP6(name, type1, arg1, type2, arg2,                     \
    type3, arg3, type4, arg4, type5, arg5, type6, arg6)                 \
        DTRACE_PROBE6(__tcp_##name, type1, arg1, type2, arg2,           \
            type3, arg3, type4, arg4, type5, arg5, type6, arg6);

#define DTRACE_UDP(name)                                                \
        DTRACE_PROBE(__udp_##name);

#define DTRACE_UDP1(name, type1, arg1)                                  \
        DTRACE_PROBE1(__udp_##name, type1, arg1);

#define DTRACE_UDP2(name, type1, arg1, type2, arg2)                     \
        DTRACE_PROBE2(__udp_##name, type1, arg1, type2, arg2);

#define DTRACE_UDP3(name, type1, arg1, type2, arg2, type3, arg3)        \
        DTRACE_PROBE3(__udp_##name, type1, arg1, type2, arg2, type3, arg3);

#define DTRACE_UDP4(name, type1, arg1, type2, arg2,                     \
    type3, arg3, type4, arg4)                                           \
        DTRACE_PROBE4(__udp_##name, type1, arg1, type2, arg2,           \
            type3, arg3, type4, arg4);

#define DTRACE_UDP5(name, type1, arg1, type2, arg2,                     \
    type3, arg3, type4, arg4, type5, arg5)                              \
        DTRACE_PROBE5(__udp_##name, type1, arg1, type2, arg2,           \
            type3, arg3, type4, arg4, type5, arg5);
```



```
#define DTRACE_DCCP(name)                                               \
        DTRACE_PROBE(__dccp_##name);

#define DTRACE_DCCP1(name, type1, arg1)                                 \
        DTRACE_PROBE1(__dccp_##name, type1, arg1);

#define DTRACE_DCCP2(name, type1, arg1, type2, arg2)                    \
        DTRACE_PROBE2(__dccp_##name, type1, arg1, type2, arg2);

#define DTRACE_DCCP3(name, type1, arg1, type2, arg2, type3, arg3)       \
        DTRACE_PROBE3(__dccp_##name, type1, arg1, type2, arg2, type3, arg3);

#define DTRACE_DCCP4(name, type1, arg1, type2, arg2,                    \
    type3, arg3, type4, arg4)                                           \
        DTRACE_PROBE4(__dccp_##name, type1, arg1, type2, arg2,          \
            type3, arg3, type4, arg4);

#define DTRACE_DCCP5(name, type1, arg1, type2, arg2,                    \
    type3, arg3, type4, arg4, type5, arg5)                              \
        DTRACE_PROBE5(__dccp_##name, type1, arg1, type2, arg2,          \
            type3, arg3, type4, arg4, type5, arg5);

#define DTRACE_DCCP6(name, type1, arg1, type2, arg2,                    \
    type3, arg3, type4, arg4, type5, arg5, type6, arg6)                 \
        DTRACE_PROBE6(__dccp_##name, type1, arg1, type2, arg2,          \
            type3, arg3, type4, arg4, type5, arg5, type6, arg6);

#endif /* ! codereview */

#define DTRACE_SYSEVENT2(name, type1, arg1, type2, arg2)                \
        DTRACE_PROBE2(__sysevent_##name, type1, arg1, type2, arg2);

#define DTRACE_XPV(name)                                                \
        DTRACE_PROBE(__xpv_##name);

#define DTRACE_XPV1(name, type1, arg1)                                  \
        DTRACE_PROBE1(__xpv_##name, type1, arg1);

#define DTRACE_XPV2(name, type1, arg1, type2, arg2)                     \
        DTRACE_PROBE2(__xpv_##name, type1, arg1, type2, arg2);

#define DTRACE_XPV3(name, type1, arg1, type2, arg2, type3, arg3)        \
        DTRACE_PROBE3(__xpv_##name, type1, arg1, type2, arg2, type3, arg3);

#define DTRACE_XPV4(name, type1, arg1, type2, arg2, type3, arg3,        \
            type4, arg4)                                                \
        DTRACE_PROBE4(__xpv_##name, type1, arg1, type2, arg2,           \
            type3, arg3, type4, arg4);

#define DTRACE_FC_1(name, type1, arg1) \
        DTRACE_PROBE1(__fc_##name, type1, arg1);

#define DTRACE_FC_2(name, type1, arg1, type2, arg2) \
        DTRACE_PROBE2(__fc_##name, type1, arg1, type2, arg2);

#define DTRACE_FC_3(name, type1, arg1, type2, arg2, type3, arg3) \
        DTRACE_PROBE3(__fc_##name, type1, arg1, type2, arg2, type3, arg3);

#define DTRACE_FC_4(name, type1, arg1, type2, arg2, type3, arg3, type4, arg4) \
        DTRACE_PROBE4(__fc_##name, type1, arg1, type2, arg2, type3, arg3, \
            type4, arg4);

#define DTRACE_FC_5(name, type1, arg1, type2, arg2, type3, arg3,        \
            type4, arg4, type5, arg5)                                   \
        DTRACE_PROBE5(__fc_##name, type1, arg1, type2, arg2, type3, arg3, \
            type4, arg4, type5, arg5);
```

```
399 #define DTRACE_SRP_1(name, type1, arg1)                                 \
400         DTRACE_PROBE1(__srp_##name, type1, arg1);

402 #define DTRACE_SRP_2(name, type1, arg1, type2, arg2)                    \
403         DTRACE_PROBE2(__srp_##name, type1, arg1, type2, arg2);

405 #define DTRACE_SRP_3(name, type1, arg1, type2, arg2, type3, arg3)       \
406         DTRACE_PROBE3(__srp_##name, type1, arg1, type2, arg2, type3, arg3);

408 #define DTRACE_SRP_4(name, type1, arg1, type2, arg2, type3, arg3,       \
409             type4, arg4)                                                \
410         DTRACE_PROBE4(__srp_##name, type1, arg1, type2, arg2,           \
411             type3, arg3, type4, arg4);

413 #define DTRACE_SRP_5(name, type1, arg1, type2, arg2, type3, arg3,       \
414             type4, arg4, type5, arg5)                                   \
415         DTRACE_PROBE5(__srp_##name, type1, arg1, type2, arg2,           \
416             type3, arg3, type4, arg4, type5, arg5);

418 #define DTRACE_SRP_6(name, type1, arg1, type2, arg2, type3, arg3,       \
419             type4, arg4, type5, arg5, type6, arg6)                      \
420         DTRACE_PROBE6(__srp_##name, type1, arg1, type2, arg2,           \
421             type3, arg3, type4, arg4, type5, arg5, type6, arg6);

423 #define DTRACE_SRP_7(name, type1, arg1, type2, arg2, type3, arg3,       \
424             type4, arg4, type5, arg5, type6, arg6, type7, arg7)         \
425         DTRACE_PROBE7(__srp_##name, type1, arg1, type2, arg2,           \
426             type3, arg3, type4, arg4, type5, arg5, type6, arg6, type7, arg7);

428 #define DTRACE_SRP_8(name, type1, arg1, type2, arg2, type3, arg3,       \
429             type4, arg4, type5, arg5, type6, arg6, type7, arg7, type8, arg8) \
430         DTRACE_PROBE8(__srp_##name, type1, arg1, type2, arg2,           \
431             type3, arg3, type4, arg4, type5, arg5, type6, arg6,         \
432             type7, arg7, type8, arg8);

434 #endif /* _KERNEL */

436 extern const char *sdt_prefix;

438 typedef struct sdt_probedesc {
439         char                    *sdpd_name;     /* name of this probe */
440         unsigned long           sdpd_offset;    /* offset of call in text */
441         struct sdt_probedesc    *sdpd_next;     /* next static probe */
442 } sdt_probedesc_t;

444 #ifdef  __cplusplus
445 }
446 #endif

448 #endif  /* _SYS_SDT_H */
```

```
**********************************************************
   16840 Mon Jul  9 14:38:23 2012
new/usr/src/uts/intel/Makefile.intel.shared
dccp: starting module template
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

# Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.

# Copyright 2011 Nexenta Systems, Inc.  All rights reserved.

#
#       This makefile contains the common definitions for all intel
#       implementation architecture independent modules.
#


#
#       Machine type (implementation architecture):
#
PLATFORM         = i86pc

#
#       Everybody needs to know how to build modstubs.o and to locate unix.o.
#       Note that unix.o must currently be selected from among the possible
#       "implementation architectures". Note further, that unix.o is only
#       used as an optional error check for undefines so (theoretically)
#       any "implementation architectures" could be used. We choose i86pc
#       because it is the reference port.
#
UNIX_DIR         = $(UTSBASE)/i86pc/unix
GENLIB_DIR       = $(UTSBASE)/intel/genunix
IPDRV_DIR        = $(UTSBASE)/intel/ip
MODSTUBS_DIR     = $(UNIX_DIR)
DSF_DIR          = $(UTSBASE)/$(PLATFORM)/genassym
LINTS_DIR        = $(OBJS_DIR)
LINT_LIB_DIR     = $(UTSBASE)/intel/lint-libs/$(OBJS_DIR)

UNIX_O           = $(UNIX_DIR)/$(OBJS_DIR)/unix.o
GENLIB           = $(GENLIB_DIR)/$(OBJS_DIR)/libgenunix.so
MODSTUBS_O       = $(MODSTUBS_DIR)/$(OBJS_DIR)/modstubs.o
LINT_LIB         = $(UTSBASE)/i86pc/lint-libs/$(OBJS_DIR)/llib-lunix.ln
GEN_LINT_LIB     = $(UTSBASE)/intel/lint-libs/$(OBJS_DIR)/llib-lgenunix.ln

#
#       Include the makefiles which define build rule templates, the
#       collection of files per module, and a few specific flags. Note
#       that order is significant, just as with an include path. The
```



```
#       first build rule template which matches the files name will be
#       used. By including these in order from most machine dependent
#       to most machine independent, we allow a machine dependent file
#       to be used in preference over a machine independent version
#       (Such as a machine specific optimization, which preserves the
#       interfaces.)
#
include $(UTSTREE)/intel/Makefile.files
include $(UTSTREE)/common/Makefile.files

#
# ----- TRANSITIONAL SECTION --------------------------------------------------
#


#
#       Not everything which *should* be a module is a module yet. The
#       following is a list of such objects which are currently part of
#       genunix but which might someday become kmods.  This must be
#       defined before we include Makefile.uts, or else genunix's build
#       won't be as parallel as we might like.
#
NOT_YET_KMODS    = $(OLDPTY_OBJS) $(PTY_OBJS) $(VCONS_CONF_OBJS) $(MOD_OBJS)

#
# ----- END OF TRANSITIONAL SECTION -------------------------------------------
#
#       Include machine independent rules. Note that this does not imply
#       that the resulting module from rules in Makefile.uts is machine
#       independent. Only that the build rules are machine independent.
#
include $(UTSBASE)/Makefile.uts

#
#       The following must be defined for all implementations:
#
MODSTUBS               = $(UTSBASE)/intel/ia32/ml/modstubs.s

#
#       Define supported builds
#
DEF_BUILDS             = $(DEF_BUILDS64) $(DEF_BUILDS32)
ALL_BUILDS             = $(ALL_BUILDS64) $(ALL_BUILDS32)

#
#       x86 or amd64 inline templates
#
INLINES_32             = $(UTSBASE)/intel/ia32/ml/ia32.il
INLINES_64             = $(UTSBASE)/intel/amd64/ml/amd64.il
INLINES                += $(INLINES_$(CLASS))

#
#       kernel-specific optimizations; override default in Makefile.master
#

CFLAGS_XARCH_32        = $(i386_CFLAGS)
CFLAGS_XARCH_64        = $(amd64_CFLAGS)
CFLAGS_XARCH           = $(CFLAGS_XARCH_$(CLASS))

COPTFLAG_32            = $(COPTFLAG)
COPTFLAG_64            = $(COPTFLAG64)
COPTIMIZE              = $(COPTFLAG_$(CLASS))

CFLAGS                 = $(CFLAGS_XARCH)
CFLAGS                 += $(COPTIMIZE)
CFLAGS                 += $(INLINES) -D_ASM_INLINES
CFLAGS                 += $(CCMODE)
```

```
128 CFLAGS                  += $(SPACEFLAG)
129 CFLAGS                  += $(CCUNBOUND)
130 CFLAGS                  += $(CFLAGS_uts)
131 CFLAGS                  += -xstrconst

133 ASFLAGS_XARCH_32        = $(i386_ASFLAGS)
134 ASFLAGS_XARCH_64        = $(amd64_ASFLAGS)
135 ASFLAGS_XARCH           = $(ASFLAGS_XARCH_$(CLASS))

137 ASFLAGS                  += $(ASFLAGS_XARCH)

139 #
140 #       Define the base directory for installation.
141 #
142 BASE_INS_DIR    = $(ROOT)

144 #
145 #       Debugging level
146 #
147 #       Special knowledge of which special debugging options affect which
148 #       file is used to optimize the build if these flags are changed.
149 #
150 DEBUG_DEFS_OBJ32        =
151 DEBUG_DEFS_DBG32        = -DDEBUG
152 DEBUG_DEFS_OBJ64        =
153 DEBUG_DEFS_DBG64        = -DDEBUG
154 DEBUG_DEFS              = $(DEBUG_DEFS_$(BUILD_TYPE))

156 DEBUG_COND_OBJ32        :sh = echo \\043
157 DEBUG_COND_DBG32        =
158 DEBUG_COND_OBJ64        :sh = echo \\043
159 DEBUG_COND_DBG64        =
160 IF_DEBUG_OBJ            = $(DEBUG_COND_$(BUILD_TYPE))$(OBJS_DIR)/

162 $(IF_DEBUG_OBJ)syscall.o        :=      DEBUG_DEFS      += -DSYSCALLTRACE
163 $(IF_DEBUG_OBJ)clock.o          :=      DEBUG_DEFS      += -DKSLICE=1

165 #
166 #       Collect the preprocessor definitions to be associated with *all*
167 #       files.
168 #
169 ALL_DEFS         = $(DEBUG_DEFS) $(OPTION_DEFS)

171 #
172 #       The kernels modules which are "implementation architecture"
173 #       specific for this machine are enumerated below. Note that most
174 #       of these modules must exist (in one form or another) for each
175 #       architecture.
176 #
177 #       Common Drivers (usually pseudo drivers) (/kernel/drv)
178 #       DRV_KMODS are built both 32-bit and 64-bit
179 #       DRV_KMODS_32 are built only 32-bit
180 #       DRV_KMODS_64 are built only 64-bit
181 #
182 DRV_KMODS       += aac
183 DRV_KMODS       += aggr
184 DRV_KMODS       += ahci
185 DRV_KMODS       += amd64_gart
186 DRV_KMODS       += amr
187 DRV_KMODS       += agpgart
188 DRV_KMODS       += srn
189 DRV_KMODS       += agptarget
190 DRV_KMODS       += arn
191 DRV_KMODS       += arp
192 DRV_KMODS       += asy
193 DRV_KMODS       += ata
```



```
194 DRV_KMODS       += ath
195 DRV_KMODS       += atu
196 DRV_KMODS       += audio
197 DRV_KMODS       += audio1575
198 DRV_KMODS       += audio810
199 DRV_KMODS       += audiocmi
200 DRV_KMODS       += audiocmihd
201 DRV_KMODS       += audioemu10k
202 DRV_KMODS       += audioens
203 DRV_KMODS       += audiohd
204 DRV_KMODS       += audioixp
205 DRV_KMODS       += audiols
206 DRV_KMODS       += audiop16x
207 DRV_KMODS       += audiopci
208 DRV_KMODS       += audiosolo
209 DRV_KMODS       += audiots
210 DRV_KMODS       += audiovia823x
211 DRV_KMODS_32    += audiovia97
212 DRV_KMODS       += bl
213 DRV_KMODS       += blkdev
214 DRV_KMODS       += bge
215 DRV_KMODS       += bofi
216 DRV_KMODS       += bpf
217 DRV_KMODS       += bridge
218 DRV_KMODS       += bscbus
219 DRV_KMODS       += bscv
220 DRV_KMODS       += chxge
221 DRV_KMODS       += ntxn
222 DRV_KMODS       += myri10ge
223 DRV_KMODS       += clone
224 DRV_KMODS       += cmdk
225 DRV_KMODS       += cn
226 DRV_KMODS       += conskbd
227 DRV_KMODS       += consms
228 DRV_KMODS       += cpuid
229 DRV_KMODS       += cpunex
230 DRV_KMODS       += crypto
231 DRV_KMODS       += cryptoadm
232 DRV_KMODS       += dca
233 DRV_KMODS       += dccp
234 DRV_KMODS       += dccp6
235 #endif /* ! codereview */
236 DRV_KMODS       += devinfo
237 DRV_KMODS       += dld
238 DRV_KMODS       += dlpistub
239 DRV_KMODS_32    += dnet
240 DRV_KMODS       += dump
241 DRV_KMODS       += ecpp
242 DRV_KMODS       += emlxs
243 DRV_KMODS       += fd
244 DRV_KMODS       += fdc
245 DRV_KMODS       += fm
246 DRV_KMODS       += fssnap
247 DRV_KMODS       += hxge
248 DRV_KMODS       += i8042
249 DRV_KMODS       += i915
250 DRV_KMODS       += icmp
251 DRV_KMODS       += icmp6
252 DRV_KMODS       += intel_nb5000
253 DRV_KMODS       += intel_nhm
254 DRV_KMODS       += ip
255 DRV_KMODS       += ip6
256 DRV_KMODS       += ipf
257 DRV_KMODS       += ipnet
258 DRV_KMODS       += ippctl
259 DRV_KMODS       += ipsecah
```

```
DRV_KMODS       += ipsecesp
DRV_KMODS       += ipw
DRV_KMODS       += iwh
DRV_KMODS       += iwi
DRV_KMODS       += iwk
DRV_KMODS       += iwp
DRV_KMODS       += iwscn
DRV_KMODS       += kb8042
DRV_KMODS       += keysock
DRV_KMODS       += kssl
DRV_KMODS       += kstat
DRV_KMODS       += ksyms
DRV_KMODS       += kmdb
DRV_KMODS       += llc1
DRV_KMODS       += lofi
DRV_KMODS       += log
DRV_KMODS       += logindmux
DRV_KMODS       += mega_sas
DRV_KMODS       += mc-amd
DRV_KMODS       += mm
DRV_KMODS       += mouse8042
DRV_KMODS       += mpt_sas
DRV_KMODS       += mr_sas
DRV_KMODS       += mwl
DRV_KMODS       += nca
DRV_KMODS       += nsmb
DRV_KMODS       += nulldriver
DRV_KMODS       += nv_sata
DRV_KMODS       += nxge
DRV_KMODS       += oce
DRV_KMODS       += openeepr
DRV_KMODS       += pci_pci
DRV_KMODS       += pcic
DRV_KMODS       += pcieb
DRV_KMODS       += physmem
DRV_KMODS       += pcan
DRV_KMODS       += pcwl
DRV_KMODS       += pit_beep
DRV_KMODS       += pm
DRV_KMODS       += poll
DRV_KMODS       += pool
DRV_KMODS       += power
DRV_KMODS       += pseudo
DRV_KMODS       += ptc
DRV_KMODS       += ptm
DRV_KMODS       += pts
DRV_KMODS       += ptsl
DRV_KMODS       += qlge
DRV_KMODS       += radeon
DRV_KMODS       += ral
DRV_KMODS       += ramdisk
DRV_KMODS       += random
DRV_KMODS       += rds
DRV_KMODS       += rdsv3
DRV_KMODS       += rpcib
DRV_KMODS       += rsm
DRV_KMODS       += rts
DRV_KMODS       += rtw
DRV_KMODS       += rum
DRV_KMODS       += rwd
DRV_KMODS       += rwn
DRV_KMODS       += sad
DRV_KMODS       += sd
DRV_KMODS       += sdhost
DRV_KMODS       += sgen
DRV_KMODS       += si3124
```



```
DRV_KMODS       += smbios
DRV_KMODS       += softmac
DRV_KMODS       += spdsock
DRV_KMODS       += smbsrv
DRV_KMODS       += smp
DRV_KMODS       += sppp
DRV_KMODS       += sppptun
DRV_KMODS       += srpt
DRV_KMODS       += st
DRV_KMODS       += sy
DRV_KMODS       += sysevent
DRV_KMODS       += sysmsg
DRV_KMODS       += tcp
DRV_KMODS       += tcp6
DRV_KMODS       += tl
DRV_KMODS       += tnf
DRV_KMODS       += tpm
DRV_KMODS       += trill
DRV_KMODS       += udp
DRV_KMODS       += udp6
DRV_KMODS       += ucode
DRV_KMODS       += ural
DRV_KMODS       += uath
DRV_KMODS       += urtw
DRV_KMODS       += vgatext
DRV_KMODS       += heci
DRV_KMODS       += vnic
DRV_KMODS       += vscan
DRV_KMODS       += wc
DRV_KMODS       += winlock
DRV_KMODS       += wpi
DRV_KMODS       += xge
DRV_KMODS       += yge
DRV_KMODS       += zcons
DRV_KMODS       += zyd
DRV_KMODS       += simnet
DRV_KMODS       += stmf
DRV_KMODS       += stmf_sbd
DRV_KMODS       += fct
DRV_KMODS       += fcoe
DRV_KMODS       += fcoet
DRV_KMODS       += fcoei
DRV_KMODS       += qlt
DRV_KMODS       += iscsit
DRV_KMODS       += pppt
DRV_KMODS       += ncall nsctl sdbc nskern sv
DRV_KMODS       += ii rdc rdcsrv rdcstub
DRV_KMODS       += iptun

$(CLOSED_BUILD)CLOSED_DRV_KMODS         += bmc
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += glm
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += intel_nhmex
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += cpqary3
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += marvell88sx
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += bcm_sata
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += memtest
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += mpt
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += atiatom
$(CLOSED_BUILD)CLOSED_DRV_KMODS         += acpi_toshiba

#
# Common code drivers
#

DRV_KMODS       += afe
DRV_KMODS       += atge
```

```
392 DRV_KMODS       += bfe
393 DRV_KMODS       += dmfe
394 DRV_KMODS       += e1000g
395 DRV_KMODS       += efe
396 DRV_KMODS       += elxl
397 DRV_KMODS       += hme
398 DRV_KMODS       += mxfe
399 DRV_KMODS       += nge
400 DRV_KMODS       += pcn
401 DRV_KMODS       += rge
402 DRV_KMODS       += rtls
403 DRV_KMODS       += sfe
404 DRV_KMODS       += amd8111s
405 DRV_KMODS       += igb
406 DRV_KMODS       += iprb
407 DRV_KMODS       += ixgbe
408 DRV_KMODS       += vr
409 $(CLOSED_BUILD)CLOSED_DRV_KMODS += ixgb

411 #
412 #       DTrace and DTrace Providers
413 #
414 DRV_KMODS       += dtrace
415 DRV_KMODS       += fbt
416 DRV_KMODS       += lockstat
417 DRV_KMODS       += profile
418 DRV_KMODS       += sdt
419 DRV_KMODS       += systrace
420 DRV_KMODS       += fasttrap
421 DRV_KMODS       += dcpc

423 #
424 #       I/O framework test drivers
425 #
426 DRV_KMODS       += pshot
427 DRV_KMODS       += gen_drv
428 DRV_KMODS       += tvhci tphci tclient
429 DRV_KMODS       += emul64

431 #
432 #       Machine Specific Driver Modules (/kernel/drv):
433 #
434 DRV_KMODS       += options
435 DRV_KMODS       += scsi_vhci
436 DRV_KMODS       += pmcs
437 DRV_KMODS       += pmcs8001fw
438 DRV_KMODS       += arcmsr
439 DRV_KMODS       += fcp
440 DRV_KMODS       += fcip
441 DRV_KMODS       += fcsm
442 DRV_KMODS       += fp
443 DRV_KMODS       += qlc
444 DRV_KMODS       += iscsi

446 #
447 #       PCMCIA specific module(s)
448 #
449 DRV_KMODS       += pcs
450 DRV_KMODS       += pcata
451 MISC_KMODS      += cardbus
452 $(CLOSED_BUILD)CLOSED_DRV_KMODS += pcser

454 #
455 #       SCSI Enclosure Services driver
456 #
457 DRV_KMODS       += ses
```



```
459 #
460 #       USB specific modules
461 #
462 DRV_KMODS       += hid
463 DRV_KMODS       += hwarc hwahc
464 DRV_KMODS       += hubd
465 DRV_KMODS       += uhci
466 DRV_KMODS       += ehci
467 DRV_KMODS       += ohci
468 DRV_KMODS       += usb_mid
469 DRV_KMODS       += usb_ia
470 DRV_KMODS       += scsa2usb
471 DRV_KMODS       += usbprn
472 DRV_KMODS       += ugen
473 DRV_KMODS       += usbser
474 DRV_KMODS       += usbsacm
475 DRV_KMODS       += usbsksp
476 DRV_KMODS       += usbsprl
477 DRV_KMODS       += usb_ac
478 DRV_KMODS       += usb_as
479 DRV_KMODS       += usbskel
480 DRV_KMODS       += usbvc
481 DRV_KMODS       += usbftdi
482 DRV_KMODS       += wusb_df
483 DRV_KMODS       += wusb_ca
484 DRV_KMODS       += usbecm

486 $(CLOSED_BUILD)CLOSED_DRV_KMODS += usbser_edge

488 #
489 #       1394 modules
490 #
491 MISC_KMODS      += s1394 sbp2
492 DRV_KMODS       += hci1394 scsa1394
493 DRV_KMODS       += av1394
494 DRV_KMODS       += dcam1394

496 #
497 #       InfiniBand pseudo drivers
498 #
499 DRV_KMODS       += ib ibp eibnx eoib rdsib sdp iser daplt hermon tavor sol_ucma
500 DRV_KMODS       += sol_umad

502 #
503 #       LVM modules
504 #
505 DRV_KMODS       += md
506 MISC_KMODS      += md_stripe md_hotspares md_mirror md_raid md_trans md_notify
507 MISC_KMODS      += md_sp

509 #
510 #       Brand modules
511 #
512 BRAND_KMODS     += sn1_brand s10_brand

514 #
515 #       Exec Class Modules (/kernel/exec):
516 #
517 EXEC_KMODS      += elfexec intpexec shbinexec javaexec

519 #
520 #       Scheduling Class Modules (/kernel/sched):
521 #
522 SCHED_KMODS     += IA RT TS RT_DPTBL TS_DPTBL FSS FX FX_DPTBL SDC
```

```
524 #
525 #       File System Modules (/kernel/fs):
526 #
527 FS_KMODS        += autofs cachefs ctfs dcfs dev devfs fdfs fifofs hsfs lofs
528 FS_KMODS        += mntfs namefs nfs objfs zfs zut
529 FS_KMODS        += pcfs procfs sockfs specfs tmpfs udfs ufs sharefs
530 FS_KMODS        += smbfs

532 #
533 #       Streams Modules (/kernel/strmod):
534 #
535 STRMOD_KMODS    += bufmod connld dedump ldterm pckt pfmod pipemod
536 STRMOD_KMODS    += ptem redirmod rpcmod rlmod telmod timod
537 STRMOD_KMODS    += spppasyn spppcomp
538 STRMOD_KMODS    += tirdwr ttcompat
539 STRMOD_KMODS    += usbkbm
540 STRMOD_KMODS    += usbms
541 STRMOD_KMODS    += usbwcm
542 STRMOD_KMODS    += usb_ah
543 STRMOD_KMODS    += drcompat
544 STRMOD_KMODS    += cryptmod
545 STRMOD_KMODS    += vuid2ps2
546 STRMOD_KMODS    += vuid3ps2
547 STRMOD_KMODS    += vuidm3p
548 STRMOD_KMODS    += vuidm4p
549 STRMOD_KMODS    += vuidm5p

551 #
552 #       'System' Modules (/kernel/sys):
553 #
554 SYS_KMODS       += c2audit
555 SYS_KMODS       += doorfs
556 SYS_KMODS       += exacctsys
557 SYS_KMODS       += inst_sync
558 SYS_KMODS       += kaio
559 SYS_KMODS       += msgsys
560 SYS_KMODS       += pipe
561 SYS_KMODS       += portfs
562 SYS_KMODS       += pset
563 SYS_KMODS       += semsys
564 SYS_KMODS       += shmsys
565 SYS_KMODS       += sysacct
566 SYS_KMODS       += acctctl

568 #
569 #       'Misc' Modules (/kernel/misc)
570 #       MISC_KMODS are built both 32-bit and 64-bit
571 #       MISC_KMODS_32 are built only 32-bit
572 #       MISC_KMODS_64 are built only 64-bit
573 #
574 MISC_KMODS      += ac97
575 MISC_KMODS      += acpica
576 MISC_KMODS      += agpmaster
577 MISC_KMODS      += bignum
578 MISC_KMODS      += bootdev
579 MISC_KMODS      += busra
580 MISC_KMODS      += cmlb
581 MISC_KMODS      += consconfig
582 MISC_KMODS      += ctf
583 MISC_KMODS      += dadk
584 MISC_KMODS      += dcopy
585 MISC_KMODS      += dls
586 MISC_KMODS      += drm
587 MISC_KMODS      += fssnap_if
588 MISC_KMODS      += gda
589 MISC_KMODS      += gld
```



```
590 MISC_KMODS      += hidparser
591 MISC_KMODS      += hook
592 MISC_KMODS      += hpcsvc
593 MISC_KMODS      += ibcm
594 MISC_KMODS      += ibdm
595 MISC_KMODS      += ibdma
596 MISC_KMODS      += ibmf
597 MISC_KMODS      += ibtl
598 MISC_KMODS      += idm
599 MISC_KMODS      += idmap
600 MISC_KMODS      += iommulib
601 MISC_KMODS      += ipc
602 MISC_KMODS      += kbtrans
603 MISC_KMODS      += kcf
604 MISC_KMODS      += kgssapi
605 MISC_KMODS      += kmech_dummy
606 MISC_KMODS      += kmech_krb5
607 MISC_KMODS      += ksocket
608 MISC_KMODS      += mac
609 MISC_KMODS      += mii
610 MISC_KMODS      += mwlfw
611 MISC_KMODS      += net80211
612 MISC_KMODS      += nfs_dlboot
613 MISC_KMODS      += nfssrv
614 MISC_KMODS      += neti
615 MISC_KMODS      += pci_autoconfig
616 MISC_KMODS      += pcicfg
617 MISC_KMODS      += pcihp
618 MISC_KMODS      += pcmcia
619 MISC_KMODS      += rpcsec
620 MISC_KMODS      += rpcsec_gss
621 MISC_KMODS      += rsmops
622 MISC_KMODS      += sata
623 MISC_KMODS      += scsi
624 MISC_KMODS      += sda
625 MISC_KMODS      += sol_ofs
626 MISC_KMODS      += spuni
627 MISC_KMODS      += strategy
628 MISC_KMODS      += strplumb
629 MISC_KMODS      += tem
630 MISC_KMODS      += tlimod
631 MISC_KMODS      += usba usba10 usbs49_fw
632 MISC_KMODS      += scsi_vhci_f_sym_hds
633 MISC_KMODS      += scsi_vhci_f_sym
634 MISC_KMODS      += scsi_vhci_f_tpgs
635 MISC_KMODS      += scsi_vhci_f_asym_sun
636 MISC_KMODS      += scsi_vhci_f_tape
637 MISC_KMODS      += scsi_vhci_f_tpgs_tape
638 MISC_KMODS      += fctl
639 MISC_KMODS      += emlxs_fw
640 MISC_KMODS      += qlc_fw_2200
641 MISC_KMODS      += qlc_fw_2300
642 MISC_KMODS      += qlc_fw_2400
643 MISC_KMODS      += qlc_fw_2500
644 MISC_KMODS      += qlc_fw_6322
645 MISC_KMODS      += qlc_fw_8100
646 MISC_KMODS      += hwa1480_fw
647 MISC_KMODS      += uathfw
648 MISC_KMODS      += uwba

650 $(CLOSED_BUILD)CLOSED_MISC_KMODS        += klmmod klmops
651 $(CLOSED_BUILD)CLOSED_MISC_KMODS        += scsi_vhci_f_asym_lsi
652 $(CLOSED_BUILD)CLOSED_MISC_KMODS        += scsi_vhci_f_asym_emc
653 $(CLOSED_BUILD)CLOSED_MISC_KMODS        += scsi_vhci_f_sym_emc

655 #
```

```
#       Software Cryptographic Providers (/kernel/crypto):
#
CRYPTO_KMODS    += aes
CRYPTO_KMODS    += arcfour
CRYPTO_KMODS    += blowfish
CRYPTO_KMODS    += des
CRYPTO_KMODS    += ecc
CRYPTO_KMODS    += md4
CRYPTO_KMODS    += md5
CRYPTO_KMODS    += rsa
CRYPTO_KMODS    += sha1
CRYPTO_KMODS    += sha2
CRYPTO_KMODS    += swrand

#
#       IP Policy Modules (/kernel/ipp)
#
IPP_KMODS       += dlcosmk
IPP_KMODS       += flowacct
IPP_KMODS       += ipgpc
IPP_KMODS       += dscpmk
IPP_KMODS       += tokenmt
IPP_KMODS       += tswtclmt

#
#       generic-unix module (/kernel/genunix):
#
GENUNIX_KMODS   += genunix

#
#       SVVS Testing Modules (/kernel/strmod):
#
#       These are streams and driver modules which are not to be
#       delivered with a released system. However, during development
#       it is convenient to build and install the SVVS kernel modules.
#
SVVS_KMODS      += lmodb lmode lmodr lmodt svvslo tidg tivc tmux

$(CLOSED_BUILD)SVVS             += svvs

#
#       Modules eXcluded from the product:
#
$(CLOSED_BUILD)CLOSED_XMODS =           \
        adpu320         \
        bnx             \
        bnxe            \
        lsimega         \
        sdpib


#
#       'Dacf' Modules (/kernel/dacf):
#

#
#       Performance Counter BackEnd modules (/usr/kernel/pcbe)
#
PCBE_KMODS      += p123_pcbe p4_pcbe opteron_pcbe core_pcbe

#
#       MAC-Type Plugin Modules (/kernel/mac)
#
MAC_KMODS       += mac_6to4
MAC_KMODS       += mac_ether
MAC_KMODS       += mac_ipv4
```



```
MAC_KMODS       += mac_ipv6
MAC_KMODS       += mac_wifi
MAC_KMODS       += mac_ib

#
# socketmod (kernel/socketmod)
#
SOCKET_KMODS    += sockpfp
SOCKET_KMODS    += socksctp
SOCKET_KMODS    += socksdp
SOCKET_KMODS    += sockrds
SOCKET_KMODS    += ksslf

#
#       kiconv modules (/kernel/kiconv):
#
KICONV_KMODS    += kiconv_emea kiconv_ja kiconv_ko kiconv_sc kiconv_tc

#
#       'Dacf' Modules (/kernel/dacf):
#
DACF_KMODS      += net_dacf
```

```
**********************************************************
    2348 Mon Jul  9 14:38:23 2012
new/usr/src/uts/intel/dccp/Makefile
dccp: starting with options
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# uts/intel/dccp/Makefile
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#       This makefile drives the production of the dccp driver kernel module.
#
#       intel implementation architecture dependent
#


#
#       Path to the base of the uts directory tree (usually /usr/src/uts).
#
UTSBASE = ../..

#
#       Define the module and object file sets.
#
MODULE          = dccp
OBJECTS         = $(DCCP_OBJS:%=$(OBJS_DIR)/%)
LINTS           = $(DCCP_OBJS:%.o=$(LINTS_DIR)/%.ln)
ROOTMODULE      = $(ROOT_DRV_DIR)/$(MODULE)
ROOTLINK        = $(ROOT_STRMOD_DIR)/$(MODULE) $(ROOT_SOCK_DIR)/$(MODULE)
CONF_SRCDIR     = $(UTSBASE)/common/inet/dccp

#
# Extra for $(MODULE).check target
#
# Need to remove ipddi.o since it has non-static defines for _init etc.
IP_CHECK_OBJS   = $(IP_OBJS:ipddi.o=ip.o)
EXTRA_CHECK_OBJS = $(IP_CHECK_OBJS:%=../ip/$(OBJS_DIR)/%)

#
#       Include common rules.
#
include $(UTSBASE)/intel/Makefile.intel

#
#       Define targets
#
```



```
ALL_TARGET      = $(BINARY) $(SRC_CONFILE)
LINT_TARGET     = $(MODULE).lint
INSTALL_TARGET  = $(BINARY) $(ROOTMODULE) $(ROOTLINK) $(ROOT_CONFFILE)

#
#       depends on ip and sockfs
#
LDFLAGS         += -dy -Ndrv/ip -Nfs/sockfs

#
#       Default build targets.
#
.KEEP_STATE:

def:            $(DEF_DEPS)

all:            $(ALL_DEPS)

clean:          $(CLEAN_DEPS)

clobber:        $(CLOBBER_DEPS)

lint:           $(LINT_DEPS)

modlintlib:     $(MODLINTLIB_DEPS)

clean.lint:     $(CLEAN_LINT_DEPS)

install:        $(INSTALL_DEPS)

$(ROOTLINK):    $(ROOT_STRMOD_DIR) $(ROOT_SOCK_DIR) $(ROOTMODULE)
        -$(RM) $@; ln $(ROOTMODULE) $@

#
#       Include common targets.
#
include $(UTSBASE)/intel/Makefile.targ
#endif /* ! codereview */
```

```
**********************************************************
    2178 Mon Jul  9 14:38:23 2012
new/usr/src/uts/intel/dccp6/Makefile
dccp: clean up
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License, Version 1.0 only
# (the "License").  You may not use this file except in compliance
# with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#       This makefile drives the production of the dccp6 driver kernel module.
#
#       intel implementation architecture dependent
#


#
#       Path to the base of the uts directory tree (usually /usr/src/uts).
#
UTSBASE = ../..

#
#       Define the module and object file sets.
#
MODULE          = dccp6
OBJECTS         = $(TCP6_OBJS:%=$(OBJS_DIR)/%)
LINTS           = $(TCP6_OBJS:%.o=$(LINTS_DIR)/%.ln)
ROOTMODULE      = $(ROOT_DRV_DIR)/$(MODULE)
CONF_SRCDIR     = $(UTSBASE)/common/inet/dccp

#
# Extra for $(MODULE).check target
#
# Need to remove ipddi.o since it has non-static defines for _init etc.
IP_CHECK_OBJS   = $(IP_OBJS:ipddi.o=ip.o)
EXTRA_CHECK_OBJS = $(IP_CHECK_OBJS:%=../ip/$(OBJS_DIR)/%)

#
#       Include common rules.
#
include $(UTSBASE)/intel/Makefile.intel

#
#       Define targets
#
ALL_TARGET      = $(BINARY) $(SRC_CONFFILE)
LINT_TARGET     = $(MODULE).lint
```



```
INSTALL_TARGET  = $(BINARY) $(ROOTMODULE) $(ROOT_CONFFILE)

#
#       depends on tcp ip and ip6
#
LDFLAGS         += -dy -Ndrv/tcp -Ndrv/ip -Ndrv/ip6

#
#       Default build targets.
#
.KEEP_STATE:

def:            $(DEF_DEPS)

all:            $(ALL_DEPS)

clean:          $(CLEAN_DEPS)

clobber:        $(CLOBBER_DEPS)

lint:           $(LINT_DEPS)

modlintlib:     $(MODLINTLIB_DEPS)

clean.lint:     $(CLEAN_LINT_DEPS)

install:        $(INSTALL_DEPS)

#
#       Include common targets.
#
include $(UTSBASE)/intel/Makefile.targ
#endif /* ! codereview */
```

```
**********************************************************
    6183 Mon Jul  9 14:38:23 2012
new/usr/src/uts/intel/ip/ip.global-objs.debug64
dccp: starting module template
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2011 Nexenta Systems, Inc. All rights reserved
#

arp_m_tbl
arp_mod_info
arp_netinfo
arp_no_defense
arpinfo
cb_inet_devops
cl_inet_bind
cl_inet_checkspi
cl_inet_connect2
cl_inet_deletespi
cl_inet_disconnect
cl_inet_getspi
cl_inet_idlesa
cl_inet_ipident
cl_inet_isclusterwide
cl_inet_listen
cl_inet_unbind
cl_inet_unlisten
cl_sctp_assoc_change
cl_sctp_check_addrs
cl_sctp_connect
cl_sctp_disconnect
cl_sctp_listen
cl_sctp_unlisten
conn_drain_nthreads
dccp_acceptor_rinit
dccp_acceptor_winit
dccp_bind_fanout_size
dccp_conn_cache
dccp_fallback_sock_winit
dccp_g_t_info_ack
dccp_g_t_info_ack_v6
dccp_max_optsize
dccp_opt_arr
dccp_opt_obj
dccp_propinfo_count
```



```
dccp_propinfo_tbl
dccp_random_anon_port
dccp_rinfo
dccp_rinitv4
dccp_rinitv6
dccp_sock_winit
dccp_squeue_flag
dccp_valid_levels_arr
dccp_winfo
dccp_winit
dccpinfov4
dccpinfov6
#endif /* ! codereview */
dce_cache
default_ip6_asp_table
do_tcp_fusion
do_tcpzcopy
dohwcksum
dummy_mod_info
dummymodinfo
dummyrmodinit
dummywmodinit
eventq_queue_in
eventq_queue_nic
eventq_queue_out
fsw
gcdb_hash
gcdb_hash_size
gcdb_lock
gcgrp4_hash
gcgrp6_hash
gcgrp_hash_size
gcgrp_lock
icmp_fallback_sock_winit
icmp_frag_size_table
icmp_g_t_info_ack
icmp_ipha
icmp_max_optsize
icmp_mod_info
icmp_opt_arr
icmp_opt_obj
icmp_propinfo_tbl
icmp_valid_levels_arr
icmpinfov4
icmpinfov6
icmprinitv4
icmprinitv6
icmpwinit
ilb_conn_cache
ilb_conn_cache_timeout
ilb_conn_hash_size
ilb_conn_tcp_expiry
ilb_conn_timer_size
ilb_conn_udp_expiry
ilb_kstat_instance
ilb_kmem_flags
ilb_nat_src_hash_size
ilb_nat_src_instance
ilb_rule_hash_size
ilb_sticky_cache
ilb_sticky_hash_size
ilb_sticky_expiry
ilb_sticky_timer_size
ilb_sticky_timeout
ill_no_arena
ill_null
```

```
inet_dev_info
inet_devops
ip6_ftable_hash_size
ip6opt_ls
ip_cgtp_filter_rev
ip_conn_cache
ip_debug
ip_g_all_ones
ip_helper_stream_info
ip_helper_stream_rinit
ip_helper_stream_winit
ip_ioctl_ftbl
ip_loopback_mtu_v6plus
ip_loopback_mtuplus
ip_m_tbl
ip_max_frag_dups
ip_min_frag_prune_time
ip_minor_arena_la
ip_minor_arena_sa
ip_misc_ioctl_count
ip_misc_ioctl_table
ip_mod_info
ip_modclose_ackwait_ms
ip_ndx_ioctl_count
ip_ndx_ioctl_table
ip_poll_normal_ms
ip_poll_normal_ticks
ip_propinfo_tbl
ip_propinfo_count
ip_rput_pullups
ip_six_byte_all_ones
ip_squeue_create_callback
ip_squeue_enter
ip_squeue_fanout
ip_squeue_flag
ip_squeue_worker_wait
ip_thread_data
ip_thread_list
ip_thread_rwlock
ipcl_bind_fanout_size
ipcl_conn_hash_maxsize
ipcl_conn_hash_memfactor
ipcl_conn_hash_size
ipcl_dccp_fanout_size
#endif /* ! codereview */
ipcl_iptun_fanout_size
ipcl_raw_fanout_size
ipcl_udp_fanout_size
ipif_loopback_name
ipif_zero
ipinfov4
ipinfov6
iplrinit
iplwinit
ipmp_kstats
iprinitv4
iprinitv6
ipsec_action_cache
ipsec_hdr_pullup_needed
ipsec_pol_cache
ipsec_policy_failure_msgs
ipsec_sel_cache
ipsec_spd_hashsize
ipsec_weird_null_inbound_policy
ipv4info
ipv6_all_hosts_mcast
```



```
ipv6_all_ones
ipv6_all_rtrs_mcast
ipv6_all_v2rtrs_mcast
ipv6_all_zeros
ipv6_ll_template
ipv6_loopback
ipv6_solicited_node_mcast
ipv6_unspecified_group
ipv6info
ipwinit
ire_cache
ire_gw_secattr_cache
ire_null
ire_nv_arr
ire_nv_tbl
lcl_param_arr
mask_rnhead
max_keylen
modldrv
modlinkage
modlstrmod
multicast_encap_iphdr
nce_cache
ncec_cache
netdev_privs
prov_update_handle
radix_mask_cache
radix_node_cache
rawip_conn_cache
recvq_call
recvq_loop_cnt
req_arr
rinit_arp
rn_mkfreelist
rn_ones
rn_zeros
rt_entry_cache
rts_conn_cache
rts_g_t_info_ack
rts_max_optsize
rts_mod_info
rts_opt_arr
rts_opt_obj
rts_valid_levels_arr
rtsinfo
rtsrinit
rtswinit
sctp_asconf_default_dispatch
sctp_asconf_dispatch_tbl
sctp_conn_cache
sctp_conn_hash_size
sctp_do_reclaim
sctp_kmem_faddr_cache
sctp_kmem_ftsn_set_cache
sctp_kmem_set_cache
sctp_min_assoc_listener
sctp_opt_arr
sctp_opt_arr_size
sctp_pa_early_abort
sctp_pp_early_abort
sctp_propinfo_tbl
sctp_propinfo_count
sctp_recvq_tq_list_max
sctp_recvq_tq_task_min
sctp_recvq_tq_thr_max
sctp_recvq_tq_thr_min
```

```
sctp_sin6_null
sctpdebug
sin6_null
sin_null
skip_sctp_cksum
sock_dccp_downcalls
#endif /* ! codereview */
sock_rawip_downcalls
sock_rts_downcalls
sock_tcp_downcalls
sock_udp_downcalls
sqset_global_list
sqset_global_size
sqset_lock
squeue_cache
squeue_drain_ms
squeue_drain_ns
squeue_workerwait_ms
squeue_workerwait_tick
tcp_acceptor_rinit
tcp_acceptor_winit
tcp_conn_cache
tcp_conn_hash_size
tcp_do_reclaim
tcp_drop_ack_unsent_cnt
tcp_dummy_upcalls
tcp_early_abort
tcp_fallback_sock_winit
tcp_free_list_max_cnt
tcp_g_kstat
tcp_g_statistics
tcp_g_t_info_ack
tcp_g_t_info_ack_v6
tcp_icmp_source_quench
tcp_init_wnd_chk
tcp_max_init_cwnd
tcp_max_optsize
tcp_min_conn_listener
tcp_notsack_blk_cache
tcp_opt_arr
tcp_opt_obj
tcp_outbound_squeue_switch
tcp_propinfo_tbl
tcp_propinfo_count
tcp_random_anon_port
tcp_random_end_ptr
tcp_random_fptr
tcp_random_lock
tcp_random_rptr
tcp_random_state
tcp_randtbl
tcp_rinfo
tcp_rinitv4
tcp_rinitv6
tcp_sock_winit
tcp_squeue_flag
tcp_squeue_wput
tcp_static_maxpsz
tcp_timercache
tcp_tx_pull_len
tcp_valid_levels_arr
tcp_winfo
tcp_winit
tcpinfov4
tcpinfov6
tli_errs
```



```
tsol_strict_error
tun_spd_hashsize
udp_bind_fanout_size
udp_conn_cache
udp_fallback_sock_winit
udp_g_t_info_ack_ipv4
udp_g_t_info_ack_ipv6
udp_lrinit
udp_lwinit
udp_max_optsize
udp_mod_info
udp_opt_arr
udp_opt_obj
udp_propinfo_tbl
udp_propinfo_count
udp_random_anon_port
udp_rinitv4
udp_rinitv6
udp_valid_levels_arr
udp_winit
udpinfov4
udpinfov6
winit_arp
nxge_cksum_workaround
```

```
**********************************************************
    6141 Mon Jul  9 14:38:24 2012
new/usr/src/uts/intel/ip/ip.global-objs.obj64
dccp: starting module template
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2011 Nexenta Systems, Inc. All rights reserved
#

arp_m_tbl
arp_mod_info
arp_netinfo
arp_no_defense
arpinfo
cb_inet_devops
cl_inet_bind
cl_inet_checkspi
cl_inet_connect2
cl_inet_deletespi
cl_inet_disconnect
cl_inet_getspi
cl_inet_idlesa
cl_inet_ipident
cl_inet_isclusterwide
cl_inet_listen
cl_inet_unbind
cl_inet_unlisten
cl_sctp_assoc_change
cl_sctp_check_addrs
cl_sctp_connect
cl_sctp_disconnect
cl_sctp_listen
cl_sctp_unlisten
conn_drain_nthreads
dccp_acceptor_rinit
dccp_acceptor_winit
dccp_bind_fanout_size
dccp_conn_cache
dccp_fallback_sock_winit
dccp_g_t_info_ack
dccp_g_t_info_ack_v6
dccp_max_optsize
dccp_opt_arr
dccp_opt_obj
dccp_propinfo_count
```



```
dccp_propinfo_tbl
dccp_random_anon_port
dccp_rinfo
dccp_rinitv4
dccp_rinitv6
dccp_sock_winit
dccp_squeue_flag
dccp_valid_levels_arr
dccp_winfo
dccp_winit
dccpinfov4
dccpinfov6
#endif /* ! codereview */
dce_cache
default_ip6_asp_table
do_tcp_fusion
do_tcpzcopy
dohwcksum
dummy_mod_info
dummymodinfo
dummyrmodinit
dummywmodinit
eventq_queue_in
eventq_queue_nic
eventq_queue_out
fsw
gcdb_hash
gcdb_hash_size
gcdb_lock
gcgrp4_hash
gcgrp6_hash
gcgrp_hash_size
gcgrp_lock
icmp_fallback_sock_winit
icmp_frag_size_table
icmp_g_t_info_ack
icmp_ipha
icmp_max_optsize
icmp_mod_info
icmp_opt_arr
icmp_opt_obj
icmp_propinfo_tbl
icmp_valid_levels_arr
icmpinfov4
icmpinfov6
icmprinitv4
icmprinitv6
icmpwinit
ilb_conn_cache
ilb_conn_cache_timeout
ilb_conn_hash_size
ilb_conn_tcp_expiry
ilb_conn_timer_size
ilb_conn_udp_expiry
ilb_kstat_instance
ilb_kmem_flags
ilb_nat_src_hash_size
ilb_nat_src_instance
ilb_rule_hash_size
ilb_sticky_cache
ilb_sticky_hash_size
ilb_sticky_expiry
ilb_sticky_timer_size
ilb_sticky_timeout
ill_no_arena
ill_null
```

```
inet_dev_info
inet_devops
ip6_ftable_hash_size
ip6opt_ls
ip_cgtp_filter_rev
ip_conn_cache
ip_debug
ip_g_all_ones
ip_helper_stream_info
ip_helper_stream_rinit
ip_helper_stream_winit
ip_ioctl_ftbl
ip_loopback_mtu_v6plus
ip_loopback_mtuplus
ip_m_tbl
ip_max_frag_dups
ip_min_frag_prune_time
ip_minor_arena_la
ip_minor_arena_sa
ip_misc_ioctl_count
ip_misc_ioctl_table
ip_mod_info
ip_modclose_ackwait_ms
ip_ndx_ioctl_count
ip_ndx_ioctl_table
ip_poll_normal_ms
ip_poll_normal_ticks
ip_propinfo_tbl
ip_propinfo_count
ip_rput_pullups
ip_six_byte_all_ones
ip_squeue_create_callback
ip_squeue_enter
ip_squeue_fanout
ip_squeue_flag
ip_squeue_worker_wait
ip_thread_data
ip_thread_list
ip_thread_rwlock
ipcl_bind_fanout_size
ipcl_conn_hash_maxsize
ipcl_conn_hash_memfactor
ipcl_conn_hash_size
ipcl_dccp_fanout_size
#endif /* ! codereview */
ipcl_iptun_fanout_size
ipcl_raw_fanout_size
ipcl_udp_fanout_size
ipif_loopback_name
ipif_zero
ipinfov4
ipinfov6
iplrinit
iplwinit
ipmp_kstats
iprinitv4
iprinitv6
ipsec_action_cache
ipsec_hdr_pullup_needed
ipsec_pol_cache
ipsec_policy_failure_msgs
ipsec_sel_cache
ipsec_spd_hashsize
ipsec_weird_null_inbound_policy
ipv4info
ipv6_all_hosts_mcast
```



```
ipv6_all_ones
ipv6_all_rtrs_mcast
ipv6_all_v2rtrs_mcast
ipv6_all_zeros
ipv6_ll_template
ipv6_loopback
ipv6_solicited_node_mcast
ipv6_unspecified_group
ipv6info
ipwinit
ire_cache
ire_gw_secattr_cache
ire_null
ire_nv_arr
ire_nv_tbl
lcl_param_arr
mask_rnhead
max_keylen
modldrv
modlinkage
modlstrmod
multicast_encap_iphdr
nce_cache
ncec_cache
netdev_privs
prov_update_handle
radix_mask_cache
radix_node_cache
rawip_conn_cache
req_arr
rinit_arp
rn_mkfreelist
rn_ones
rn_zeros
rt_entry_cache
rts_conn_cache
rts_g_t_info_ack
rts_max_optsize
rts_mod_info
rts_opt_arr
rts_opt_obj
rts_valid_levels_arr
rtsinfo
rtsrinit
rtswinit
sctp_asconf_default_dispatch
sctp_asconf_dispatch_tbl
sctp_conn_cache
sctp_conn_hash_size
sctp_do_reclaim
sctp_kmem_faddr_cache
sctp_kmem_ftsn_set_cache
sctp_kmem_set_cache
sctp_min_assoc_listener
sctp_opt_arr
sctp_opt_arr_size
sctp_pa_early_abort
sctp_pp_early_abort
sctp_propinfo_tbl
sctp_propinfo_count
sctp_recvq_tq_list_max
sctp_recvq_tq_task_min
sctp_recvq_tq_thr_max
sctp_recvq_tq_thr_min
sctp_sin6_null
sctpdebug
```

```
sin6_null
sin_null
sock_dccp_downcalls
#endif /* ! codereview */
sock_rawip_downcalls
sock_rts_downcalls
sock_tcp_downcalls
sock_udp_downcalls
sqset_global_list
sqset_global_size
sqset_lock
squeue_cache
squeue_drain_ms
squeue_drain_ns
squeue_workerwait_ms
squeue_workerwait_tick
tcp_acceptor_rinit
tcp_acceptor_winit
tcp_conn_cache
tcp_conn_hash_size
tcp_do_reclaim
tcp_drop_ack_unsent_cnt
tcp_dummy_upcalls
tcp_early_abort
tcp_fallback_sock_winit
tcp_free_list_max_cnt
tcp_g_kstat
tcp_g_statistics
tcp_g_t_info_ack
tcp_g_t_info_ack_v6
tcp_icmp_source_quench
tcp_init_wnd_chk
tcp_max_init_cwnd
tcp_max_optsize
tcp_min_conn_listener
tcp_notsack_blk_cache
tcp_opt_arr
tcp_opt_obj
tcp_outbound_squeue_switch
tcp_propinfo_tbl
tcp_propinfo_count
tcp_random_anon_port
tcp_random_end_ptr
tcp_random_fptr
tcp_random_lock
tcp_random_rptr
tcp_random_state
tcp_randtbl
tcp_rinfo
tcp_rinitv4
tcp_rinitv6
tcp_sock_winit
tcp_squeue_flag
tcp_squeue_wput
tcp_static_maxpsz
tcp_timercache
tcp_tx_pull_len
tcp_valid_levels_arr
tcp_winfo
tcp_winit
tcpinfov4
tcpinfov6
tli_errs
tsol_strict_error
tun_spd_hashsize
udp_bind_fanout_size
```



```
udp_conn_cache
udp_fallback_sock_winit
udp_g_t_info_ack_ipv4
udp_g_t_info_ack_ipv6
udp_lrinit
udp_lwinit
udp_max_optsize
udp_mod_info
udp_opt_arr
udp_opt_obj
udp_propinfo_tbl
udp_propinfo_count
udp_random_anon_port
udp_rinitv4
udp_rinitv6
udp_valid_levels_arr
udp_winit
udpinfov4
udpinfov6
winit_arp
nxge_cksum_workaround
```