
57122 Wed Jun 13 12:04:11 2012

new/usr/src/cmd/cmd-inet/usr.sbin/ipadm/ipadm.c

%B

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
637  * for all the properties for the specified object, relevant information, will
638  * be displayed. Otherwise, for the selected property set, display relevant
639  * information
640  */
641 static void
642 show_properties(void *arg, int prop_class)
643 {
644     show_prop_state_t      *statep = arg;
645     nvlist_t               *nvl = statep->sps_proplist;
646     uint_t                 proto = statep->sps_proto;
647     nvpair_t               *curr_nvp;
648     char                   *buf, *name;
649     ipadm_status_t         status;
650
651     /* allocate sufficient buffer to hold a property value */
652     if ((buf = malloc(MAXPROPVLEN)) == NULL)
653         die("insufficient memory");
654     statep->sps_propval = buf;
655
656     /* if no properties were specified, display all the properties */
657     if (nvl == NULL) {
658         (void) ipadm_walk_proptbl(proto, prop_class, show_property,
659             statep);
660     } else {
661         for (curr_nvp = nvlist_next_nvpair(nvl, NULL); curr_nvp;
662             curr_nvp = nvlist_next_nvpair(nvl, curr_nvp)) {
663             name = nvpair_name(curr_nvp);
664             status = ipadm_walk_prop(name, proto, prop_class,
665                 show_property, statep);
666             if (status == IPADM_PROP_UNKNOWN)
667                 (void) show_property(statep, name, proto);
668         }
669     }
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)
859  * given protocol or for supported protocols (IP/IPv4/IPv6/TCP/UDP/SCTP)
860  */
861 static void
862 do_show_prop(int argc, char **argv, const char *use)
863 {
864     char                   option;
865     nvlist_t               *proplist = NULL;
866     char                   *fields_str = NULL;
867     char                   *protostr;
868     show_prop_state_t     state;
869     ofmt_handle_t         ofmt;
870     ofmt_status_t         oferr;
871     uint_t                 ofmtflags = 0;
872     uint_t                 proto;
873     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     }
915
916     state.sps_proplist = proplist;
917
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;
925
926     /* handles all the errors */
927     show_properties(&state, IPADMPROP_CLASS_MODULE);
928
929     nvlist_free(proplist);
930     ofmt_close(ofmt);
931
932     if (state.sps_retstatus != IPADM_SUCCESS) {
933         ipadm_close(iph);
934         exit(EXIT_FAILURE);
935     }
936 }

```

unchanged portion omitted

new/usr/src/cmd/mdb/common/modules/ip/ip.c

1

```
*****
93952 Wed Jun 13 12:04:13 2012
```

new/usr/src/cmd/mdb/common/modules/ip/ip.c

```
%B
*****
```

_____ unchanged_portion_omitted _____

```
342 /*
343 * Generic network stack walker initialization function. It is used by all
344 * other network stack walkers.
344 * other network stack walkers.
345 */
346 int
347 ns_walk_init(mdb_walk_state_t *wsp)
348 {
349     if (mdb_layered_walk("netstack", wsp) == -1) {
350         mdb_warn("can't walk 'netstack'");
351         return (WALK_ERR);
352     }
353     return (WALK_NEXT);
354 }
```

_____ unchanged_portion_omitted _____

```
376 /*
377 * DCCP network stack walker stepping function.
378 */
379 int
380 dccp_stacks_walk_step(mdb_walk_state_t *wsp)
381 {
382     return (ns_walk_step(wsp, NS_DCCP));
383 }
```

```
385 /*
386 #endif /* ! codereview */
387 * IP network stack walker stepping function.
388 */
389 int
390 ip_stacks_walk_step(mdb_walk_state_t *wsp)
391 {
392     return (ns_walk_step(wsp, NS_IP));
393 }
```

```
395 /*
396 * TCP network stack walker stepping function.
397 */
398 int
399 tcp_stacks_walk_step(mdb_walk_state_t *wsp)
400 {
401     return (ns_walk_step(wsp, NS_TCP));
402 }
```

```
404 /*
405 * SCTP network stack walker stepping function.
406 */
407 int
408 sctp_stacks_walk_step(mdb_walk_state_t *wsp)
409 {
410     return (ns_walk_step(wsp, NS_SCTP));
411 }
```

```
413 /*
414 * UDP network stack walker stepping function.
415 */
416 int
417 udp_stacks_walk_step(mdb_walk_state_t *wsp)
418 {
```

new/usr/src/cmd/mdb/common/modules/ip/ip.c

2

```
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 to 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)
```

```

485 {
486     GElf_Sym sym;
487     list_node_t *next;

489     if (wsp->walk_addr == NULL) {
490         if (mdb_lookup_by_obj("ip", "ip_thread_list", &sym) == 0) {
491             wsp->walk_addr = sym.st_value;
492         } else {
493             mdb_warn("unable to locate ip_thread_list\n");
494             return (WALK_ERR);
495         }
496     }

498     if (mdb_vread(&next, sizeof (next),
499         wsp->walk_addr + offsetof(list_t, list_head) +
500         offsetof(list_node_t, list_next)) == -1 ||
501         next == NULL) {
502         mdb_warn("non-DEBUG image; cannot walk th_hash list\n");
503         return (WALK_ERR);
504     }

506     if (mdb_layered_walk("list", wsp) == -1) {
507         mdb_warn("can't walk 'list'");
508         return (WALK_ERR);
509     } else {
510         return (WALK_NEXT);
511     }
512 }

514 int
515 th_hash_walk_step(mdb_walk_state_t *wsp)
516 {
517     return (wsp->walk_callback(wsp->walk_addr, wsp->walk_layer,
518         wsp->walk_cbdata));
519 }

521 /*
522  * Called with walk_addr being the address of ips_ill_g_heads
523  */
524 int
525 illif_stack_walk_init(mdb_walk_state_t *wsp)
526 {
527     illif_walk_data_t *iw;

529     if (wsp->walk_addr == NULL) {
530         mdb_warn("illif_stack supports only local walks\n");
531         return (WALK_ERR);
532     }

534     iw = mdb_alloc(sizeof (illif_walk_data_t), UM_SLEEP);

536     if (mdb_vread(iw->ill_g_heads, MAX_G_HEADS * sizeof (ill_g_head_t),
537         wsp->walk_addr) == -1) {
538         mdb_warn("failed to read 'ips_ill_g_heads' at %p",
539             wsp->walk_addr);
540         mdb_free(iw, sizeof (illif_walk_data_t));
541         return (WALK_ERR);
542     }

544     iw->ill_list = 0;
545     wsp->walk_addr = (uintptr_t)iw->ill_g_heads[0].ill_g_list_head;
546     wsp->walk_data = iw;

548     return (WALK_NEXT);
549 }

```

```

551 int
552 illif_stack_walk_step(mdb_walk_state_t *wsp)
553 {
554     uintptr_t addr = wsp->walk_addr;
555     illif_walk_data_t *iw = wsp->walk_data;
556     int list = iw->ill_list;

558     if (mdb_vread(&iw->ill_if, sizeof (ill_if_t), addr) == -1) {
559         mdb_warn("failed to read ill_if_t at %p", addr);
560         return (WALK_ERR);
561     }

563     wsp->walk_addr = (uintptr_t)iw->ill_if.illif_next;

565     if (wsp->walk_addr ==
566         (uintptr_t)iw->ill_g_heads[list].ill_g_list_head) {
568         if (++list >= MAX_G_HEADS)
569             return (WALK_DONE);

571         iw->ill_list = list;
572         wsp->walk_addr =
573             (uintptr_t)iw->ill_g_heads[list].ill_g_list_head;
574         return (WALK_NEXT);
575     }

577     return (wsp->walk_callback(addr, iw, wsp->walk_cbdata));
578 }

580 void
581 illif_stack_walk_fini(mdb_walk_state_t *wsp)
582 {
583     mdb_free(wsp->walk_data, sizeof (illif_walk_data_t));
584 }

586 typedef struct illif_cbdata {
587     uint_t ill_flags;
588     uintptr_t ill_addr;
589     int ill_printlist; /* list to be printed (MAX_G_HEADS for all) */
590     boolean_t ill_printed;
591 } illif_cbdata_t;

593 static int
594 illif_cb(uintptr_t addr, const illif_walk_data_t *iw, illif_cbdata_t *id)
595 {
596     const char *version;

598     if (id->ill_printlist < MAX_G_HEADS &&
599         id->ill_printlist != iw->ill_list)
600         return (WALK_NEXT);

602     if (id->ill_flags & DCMD_ADDRSPEC && id->ill_addr != addr)
603         return (WALK_NEXT);

605     if (id->ill_flags & DCMD_PIPE_OUT) {
606         mdb_printf("%p\n", addr);
607         return (WALK_NEXT);
608     }

610     switch (iw->ill_list) {
611         case IP_V4_G_HEAD:     version = "v4"; break;
612         case IP_V6_G_HEAD:     version = "v6"; break;
613         default:                version = "??"; break;
614     }

616     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;
755
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);
764
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     }
772
773     if (nce_cb->nce_ill_name[0] != '\0' &&
774         strncmp(nce_cb->nce_ill_name, ill_name, LIFNAMSIZ) != 0)
775         return (WALK_NEXT);
776
777     if (ncec.ncec_ipversion == IPV6_VERSION) {
778
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);
785
786     } else {
787         struct in_addr nceaddr;
788
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     }
797
798     return (WALK_NEXT);
799 }
800
801 int
802 dce_walk_init(mdb_walk_state_t *wsp)
803 {
804     wsp->walk_data = (void *)wsp->walk_addr;
805
806     if (mdb_layered_walk("dce_cache", wsp) == -1) {
807         mdb_warn("can't walk 'dce_cache'");
808         return (WALK_ERR);
809     }
810
811     return (WALK_NEXT);
812 }
813
814 int

```

```

815 dce_walk_step(mdb_walk_state_t *wsp)
816 {
817     dce_t dce;
818
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     }
823
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);
827
828     return (wsp->walk_callback(wsp->walk_addr, &dce, wsp->walk_cbdata));
829 }
830
831 int
832 ire_walk_init(mdb_walk_state_t *wsp)
833 {
834     wsp->walk_data = (void *)wsp->walk_addr;
835
836     if (mdb_layered_walk("ire_cache", wsp) == -1) {
837         mdb_warn("can't walk 'ire_cache'");
838         return (WALK_ERR);
839     }
840
841     return (WALK_NEXT);
842 }
843
844 int
845 ire_walk_step(mdb_walk_state_t *wsp)
846 {
847     ire_t ire;
848
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     }
853
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);
857
858     return (wsp->walk_callback(wsp->walk_addr, &ire, wsp->walk_cbdata));
859 }
860
861 /* ARGSUSED */
862 int
863 ire_next_walk_init(mdb_walk_state_t *wsp)
864 {
865     return (WALK_NEXT);
866 }
867
868 int
869 ire_next_walk_step(mdb_walk_state_t *wsp)
870 {
871     ire_t ire;
872     int status;
873
874     if (wsp->walk_addr == NULL)
875         return (WALK_DONE);
876
877     if (mdb_vread(&ire, sizeof (ire), wsp->walk_addr) == -1) {
878         mdb_warn("can't read ire at %p", wsp->walk_addr);
879         return (WALK_ERR);
880     }

```

```

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 tmask[] = {
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 fmask[] = {
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%>\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, tmask,
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, fmask, 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%>\n"
976                  "%?s %40I\n"
977                  "%?s %40d %4d <hb> %s\n",
978                  addr, condemned ? "(C)" : "", irep->ire_setsrc_addr,
979                  irep->ire_type, tmask,
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, fmask, 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;
1017     while (sum >> 16)
1018         sum = (sum & 0xFFFF) + (sum >> 16);
1020     return (~sum);
1021 }
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 };
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
1041 static void
1042 tcp_hdr_print_options(uint8_t *opts, uint32_t opts_len)
1043 {
1044     uint8_t *endp;
1045     uint32_t len, val;
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;
1057             case TCPOPT_NOP:
1058                 mdb_printf(" NOP");
1059                 opts++;
1060                 break;
1062             case TCPOPT_MAXSEG: {
1063                 uint16_t mss;
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             }
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;
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             }
1093             opts += TCPOPT_HEADER_LEN;
1094             mdb_nhconvert(&val, opts, sizeof(val));
1095             mdb_printf(" TS_VAL=%u,", val);
1097             opts += sizeof(val);
1098             mdb_nhconvert(&val, opts, sizeof(val));
1099             mdb_printf("TS_ECHO=%u", val);
1101             opts += sizeof(val);
1102             break;
1103         }
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;
1115         case TCPOPT_SACK: {
1116             uint32_t sack_len;
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;
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                 }
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);
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(struct sctphdr *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     struct sctphdr 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         sctphdr_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, hdrrlen, 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     hdrrlen = (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 (hdrrlen == 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         hdrrlen, 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 + hdrrlen;
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 dce_masks[] = {
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(dce_masks)];
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         dce_masks);

1518     switch (dcep->dce_ipversion) {
1519     case IPV4_VERSION:
1520         mdb_printf("%<u>%?p3s %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>%?p3s %8s %8d %30N %</u>\n", addr, condemned ?
1525             "(C) : ", flagsbuf, dcep->dce_pmtu, &dcep->dce_v6addr);
1526         break;
1527     default:
1528         mdb_printf("%<u>%?p3s %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_ospace(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_ospace(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_ospace(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 %?d %?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 %?s %?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\t\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_sticky", "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_gops_t ip_gops = { ip_qinfo, ip_rnext, ip_wnext };
2021 static const mdb_modinfo_t modinfo = { MDB_API_VERSION, dcmsgs, 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_gops_install(&ip_gops, (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_gops_remove(&ip_gops, (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];
2073
2074     if (ncec->ncec_lladdr == NULL) {
2075         return ("None");
2076     }
2077
2078     if (ill->ill_net_type == IRE_IF_RESOLVER) {
2079
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 }
2097
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;
2105
2106     if (nce->nce_dlur_mp == NULL)
2107         return ("None");
2108
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 }
2139
2140 static void
2141 ncec_header(uint_t flags)
2142 {
2143     if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {
2144
2145         mdb_printf("%<u>%?s %-20s %-10s %-8s %-5s %s%</u>\n",
2146             "ADDR", "HW_ADDR", "STATE", "FLAGS", "ILL", "IP_ADDR");
2147     }
2148 }
2149
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;
2157
2158     if (mdb_getopts(argc, argv,
2159         'P', MDB_OPT_STR, &opt_P, NULL) != argc)
2160         return (DCMD_USAGE);
2161
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     }
2172
2173     if (flags & DCMD_ADDRSPEC) {
2174
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));
2185     } else {
2186         id.ncec_addr = addr;
2187         id.ncec_ipversion = ipversion;
2188         ncec_header(flags);
2189         if (mdb_walk("ncec", (mdb_walk_cb_t)ncec_cb, &id) == -1) {
2190             mdb_warn("failed to walk ncec table\n");
2191             return (DCMD_ERR);
2192         }
2193     }
2194     return (DCMD_OK);
2195 }
2196
2197
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_UNSQL_ADV, NCE_F_UNSQL_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];
2216
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 }
2222
2223 (void) mdb_readstr(ill_name, MIN(LIFNAMSIZ, ill.ill_name_length),
2224                  (uintptr_t)ill.ill_name);
2225
2226 mdb_snprintf(flagsbuf, sizeof (flagsbuf), "%hb",
2227             ncec->ncec_flags, ncec_flags);
2228
2229 if (ipversion != 0 && ncec->ncec_ipversion != ipversion)
2230     return (DCMD_OK);
2231
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 }
2248
2249 return (DCMD_OK);
2250 }
2251
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;
2257
2258     while (addr == NULL) {
2259
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;
2272
2273     kaddr4 = wsp->walk_addr + OFFSETOF(ip_stack_t, ips_ndp4);
2274     kaddr6 = wsp->walk_addr + OFFSETOF(ip_stack_t, ips_ndp6);
2275
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 }
2298
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;
2305
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 }
2321
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;
2329
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 }

```

```

2465 /* ARGSUSED */
2466 static int
2467 ncec_cb(uintptr_t addr, const ncec_walk_data_t *iw, ncec_cbdata_t *id)
2468 {
2469     ncec_t ncec;
2470
2471     if (mdb_vread(&ncec, sizeof (ncec_t), addr) == -1) {
2472         mdb_warn("failed to read ncec at %p", addr);
2473         return (WALK_NEXT);
2474     }
2475     (void) ncec_format(addr, &ncec, id->ncec_ipversion);
2476     return (WALK_NEXT);
2477 }
2478
2479 static int
2480 ill_walk_init(mdb_walk_state_t *wsp)
2481 {
2482     if (mdb_layered_walk("illif", wsp) == -1) {
2483         mdb_warn("can't walk 'illif'");
2484         return (WALK_ERR);
2485     }
2486     return (WALK_NEXT);
2487 }
2488
2489 static int
2490 ill_walk_step(mdb_walk_state_t *wsp)
2491 {
2492     ill_if_t ill_if;
2493
2494     if (mdb_vread(&ill_if, sizeof (ill_if_t), wsp->walk_addr) == -1) {
2495         mdb_warn("can't read ill_if_t at %p", wsp->walk_addr);
2496         return (WALK_ERR);
2497     }
2498     wsp->walk_addr = (uintptr_t)(wsp->walk_addr +
2499         offsetof(ill_if_t, illif_avl_by_ppa));
2500     if (mdb_pwalk("avl", wsp->walk_callback, wsp->walk_cbdata,
2501         wsp->walk_addr) == -1) {
2502         mdb_warn("can't walk 'avl'");
2503         return (WALK_ERR);
2504     }
2505     return (WALK_NEXT);
2506 }
2507
2508 /* ARGSUSED */
2509 static int
2510 ill_cb(uintptr_t addr, const ill_walk_data_t *iw, ill_cbdata_t *id)
2511 {
2512     ill_t ill;
2513
2514     if (mdb_vread(&ill, sizeof (ill_t), (uintptr_t)addr) == -1) {
2515         mdb_warn("failed to read ill at %p", addr);
2516         return (WALK_NEXT);
2517     }
2518
2519     /* If ip_stack_t is specified, skip ILLs that don't belong to it. */
2520     if (id->ill_ipst != NULL && ill.ill_ipst != id->ill_ipst)
2521         return (WALK_NEXT);
2522
2523     return (ill_format((uintptr_t)addr, &ill, id));
2524 }
2525
2526 static void
2527 ill_header(boolean_t verbose)
2528 {
2529     if (verbose) {

```

```

2531         mdb_printf("%-?s %-8s %3s %-10s %-?s %-?s %-10s%</u>\n",
2532             "ADDR", "NAME", "VER", "TYPE", "WQ", "IPST", "FLAGS");
2533         mdb_printf("%-?s %4s%4s %-?s\n",
2534             "PHYINT", "CNT", "", "GROUP");
2535         mdb_printf("%<u>%80s%</u>\n", "");
2536     } else {
2537         mdb_printf("%<u>%-?s %-8s %-3s %-10s %4s %-?s %-10s%</u>\n",
2538             "ADDR", "NAME", "VER", "TYPE", "CNT", "WQ", "FLAGS");
2539     }
2540 }
2541
2542 static int
2543 ill_format(uintptr_t addr, const void *illptr, void *ill_cb_arg)
2544 {
2545     ill_t *ill = (ill_t *)illptr;
2546     ill_cbdata_t *illcb = ill_cb_arg;
2547     boolean_t verbose = illcb->verbose;
2548     phyint_t phyi;
2549     static const mdb_bitmask_t fmask[] = {
2550         { "R",          PHYI_RUNNING,      PHYI_RUNNING },
2551         { "P",          PHYI_PROMISC,      PHYI_PROMISC },
2552         { "V",          PHYI_VIRTUAL,      PHYI_VIRTUAL },
2553         { "I",          PHYI_IPMP,         PHYI_IPMP },
2554         { "f",          PHYI_FAILED,       PHYI_FAILED },
2555         { "S",          PHYI_STANDBY,      PHYI_STANDBY },
2556         { "i",          PHYI_INACTIVE,     PHYI_INACTIVE },
2557         { "O",          PHYI_OFFLINE,      PHYI_OFFLINE },
2558         { "T",          ILLF_NOTRAILERS,   ILLF_NOTRAILERS },
2559         { "A",          ILLF_NOARP,        ILLF_NOARP },
2560         { "M",          ILLF_MULTICAST,    ILLF_MULTICAST },
2561         { "R",          ILLF_ROUTER,       ILLF_ROUTER },
2562         { "D",          ILLF_NONUD,        ILLF_NONUD },
2563         { "X",          ILLF_NORTEXCH,     ILLF_NORTEXCH },
2564         { NULL,         0,                 0 },
2565     };
2566     static const mdb_bitmask_t v_fmask[] = {
2567         { "RUNNING",    PHYI_RUNNING,      PHYI_RUNNING },
2568         { "PROMISC",    PHYI_PROMISC,      PHYI_PROMISC },
2569         { "VIRTUAL",    PHYI_VIRTUAL,      PHYI_VIRTUAL },
2570         { "IPMP",       PHYI_IPMP,         PHYI_IPMP },
2571         { "FAILED",     PHYI_FAILED,       PHYI_FAILED },
2572         { "STANDBY",    PHYI_STANDBY,      PHYI_STANDBY },
2573         { "INACTIVE",   PHYI_INACTIVE,     PHYI_INACTIVE },
2574         { "OFFLINE",    PHYI_OFFLINE,      PHYI_OFFLINE },
2575         { "NOTRAILER", ILLF_NOTRAILERS,   ILLF_NOTRAILERS },
2576         { "NOARP",      ILLF_NOARP,        ILLF_NOARP },
2577         { "MULTICAST", ILLF_MULTICAST,    ILLF_MULTICAST },
2578         { "ROUTER",     ILLF_ROUTER,       ILLF_ROUTER },
2579         { "NONUD",      ILLF_NONUD,        ILLF_NONUD },
2580         { "NORTEXCH",  ILLF_NORTEXCH,     ILLF_NORTEXCH },
2581         { NULL,         0,                 0 },
2582     };
2583     char ill_name[LIFNAMSIZ];
2584     int cnt;
2585     char *typebuf;
2586     char sbuf[DEFCOLS];
2587     int ipver = illcb->ill_ipversion;
2588
2589     if (ipver != 0) {
2590         if ((ipver == IPV4_VERSION && ill->ill_isv6) ||
2591             (ipver == IPV6_VERSION && !ill->ill_isv6)) {
2592             return (WALK_NEXT);
2593         }
2594     }
2595     if (mdb_vread(&phyi, sizeof (phyint_t),
2596         (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 %-11b\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%+--> %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 %-11b\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 fmask[] = {
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), "%11b%s",
2917 ipif->ipif_flags, fmask, 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\tTANY -> 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     "\ntips_ipcl_udp_fanout\ntips_ipcl_bind_fanout"
3126     "\ntips_ipcl_conn_fanout\ntips_ipcl_proto_fanout_v4"
3127     "\ntips_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     }
3200     return (WALK_ERR);
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(&khead, 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(&khead, 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;
3522
3523     if (wsp->walk_addr == NULL)
3524         return (WALK_ERR);
3525
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     }
3533
3534     if (sticky_walk->ilbs.ilbs_sticky_hash == NULL) {
3535         mdb_free(sticky_walk, sizeof (ilb_walk_t));
3536         return (WALK_DONE);
3537     }
3538
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;
3543
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         }
3552
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     }
3564
3565     if (st == NULL)
3566         return (WALK_DONE);
3567
3568     wsp->walk_addr = (uintptr_t)st;
3569     sticky_walk->idx = i;
3570     return (WALK_NEXT);
3571 }
3572
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;
3583
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     }
3588
3589     status = wsp->walk_callback(wsp->walk_addr, &st, wsp->walk_cbdata);
3590     if (status != WALK_NEXT)
3591         return (status);
3592
3593     sticky_walk = (ilb_walk_t *)wsp->walk_data;
3594     ilbs = &sticky_walk->ilbs;
3595     i = sticky_walk->idx;
3596
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     }
3603
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     }
3617
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         }
3628
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     }
3636
3637     if (st_next == NULL)
3638         return (WALK_DONE);
3639
3640     wsp->walk_addr = (uintptr_t)st_next;
3641     sticky_walk->idx = i;
3642     return (WALK_NEXT);
3643 }

```

```

*****
55005 Wed Jun 13 12:04:19 2012
new/usr/src/lib/libipadm/common/ipadm_prop.c
%B
*****
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 */
21 /*
22 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
23 */

25 /*
26 * This file contains routines that are used to modify/retrieve protocol or
27 * interface property values. It also holds all the supported properties for
28 * both IP interface and protocols in 'ipadm_prop_desc_t'. Following protocols
29 * are supported: IP, IPv4, IPv6, TCP, SCTP, UDP, ICMP and DCCP.
30 * are supported: IP, IPv4, IPv6, TCP, SCTP, UDP and ICMP.
31 *
32 * This file also contains walkers, which walks through the property table and
33 * calls the callback function, of the form 'ipadm_prop_wfunc_t', for every
34 * property in the table.
35 */

36 #include <unistd.h>
37 #include <errno.h>
38 #include <ctype.h>
39 #include <fcntl.h>
40 #include <strings.h>
41 #include <stdlib.h>
42 #include <netinet/in.h>
43 #include <arpa/inet.h>
44 #include <sys/socket.h>
45 #include <assert.h>
46 #include <libdlink.h>
47 #include <zone.h>
48 #include "libipadm_impl.h"
49 #include <inet/tunables.h>

51 #define IPADM_NONESTR      "none"
52 #define DEF_METRIC_VAL    0      /* default metric value */

54 #define A_CNT(arr)        (sizeof (arr) / sizeof (arr[0]))

56 static ipadm_status_t    i_ipadm_validate_if(ipadm_handle_t, const char *,
57                                         uint_t, uint_t);

59 /*
60 * Callback functions to retrieve property values from the kernel. These

```

```

61 * functions, when required, translate the values from the kernel to a format
62 * suitable for printing. For example: boolean values will be translated
63 * to on/off. They also retrieve DEFAULT, PERM and POSSIBLE values for
64 * a given property.
65 */
66 static ipadm_pd_getf_t    i_ipadm_get_prop, i_ipadm_get_ifprop_flags,
67                          i_ipadm_get_mtu, i_ipadm_get_metric,
68                          i_ipadm_get_usesrc, i_ipadm_get_forwarding,
69                          i_ipadm_get_ecnsack, i_ipadm_get_hostmodel;

71 /*
72 * Callback function to set property values. These functions translate the
73 * values to a format suitable for kernel consumption, allocates the necessary
74 * ioctl buffers and then invokes ioctl().
75 */
76 static ipadm_pd_setf_t    i_ipadm_set_prop, i_ipadm_set_mtu,
77                          i_ipadm_set_ifprop_flags,
78                          i_ipadm_set_metric, i_ipadm_set_usesrc,
79                          i_ipadm_set_forwarding, i_ipadm_set_eprivport,
80                          i_ipadm_set_ecnsack, i_ipadm_set_hostmodel;

82 /* array of protocols we support */
83 static int protocols[] = { MOD_PROTO_IP, MOD_PROTO_RAWIP,
84                           MOD_PROTO_TCP, MOD_PROTO_UDP,
85                           MOD_PROTO_SCTP, MOD_PROTO_DCCP };
86                           MOD_PROTO_SCTP };

87 /*
88 * Supported IP protocol properties.
89 */
90 static ipadm_prop_desc_t ipadm_ip_prop_table[] = {
91     { "arp", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
92       i_ipadm_set_ifprop_flags, i_ipadm_get_onoff,
93       i_ipadm_get_ifprop_flags },
94
95     { "forwarding", IPADMPROP_CLASS_MODIF, MOD_PROTO_IPV4, 0,
96       i_ipadm_set_forwarding, i_ipadm_get_onoff,
97       i_ipadm_get_forwarding },
98
99     { "metric", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
100      i_ipadm_set_metric, NULL, i_ipadm_get_metric },
101
102     { "mtu", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
103       i_ipadm_set_mtu, i_ipadm_get_mtu, i_ipadm_get_mtu },
104
105     { "exchange_routes", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
106       i_ipadm_set_ifprop_flags, i_ipadm_get_onoff,
107       i_ipadm_get_ifprop_flags },
108
109     { "usesrc", IPADMPROP_CLASS_IF, MOD_PROTO_IPV4, 0,
110       i_ipadm_set_usesrc, NULL, i_ipadm_get_usesrc },
111
112     { "ttl", IPADMPROP_CLASS_MODULE, MOD_PROTO_IPV4, 0,
113       i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },
114
115     { "forwarding", IPADMPROP_CLASS_MODIF, MOD_PROTO_IPV6, 0,
116       i_ipadm_set_forwarding, i_ipadm_get_onoff,
117       i_ipadm_get_forwarding },
118
119     { "hoplimit", IPADMPROP_CLASS_MODULE, MOD_PROTO_IPV6, 0,
120       i_ipadm_set_prop, i_ipadm_get_prop, i_ipadm_get_prop },
121
122     { "metric", IPADMPROP_CLASS_IF, MOD_PROTO_IPV6, 0,
123       i_ipadm_set_metric, NULL, i_ipadm_get_metric },
124
125     { "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     }
355
356     return (NULL);
357 }
358
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 */
383
384     return (MOD_PROTO_NONE);
385 }
386
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[MAXPROPVLEN];
398
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 = MAXPROPVLEN;
403
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     }
410
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) strncpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));
418     lifr.lifr_mtu = mtu;
419
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));
423
424     return (IPADM_SUCCESS);
425 }
426
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;
437
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     }
447
448     bzero(&lifr, sizeof (lifr));
449     (void) strncpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));
450     lifr.lifr_metric = metric;
451
452     s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);
453
454     if (ioctl(s, SIOCSLIFMETRIC, (caddr_t)&lifr) < 0)
455         return (ipadm_errno2status(errno));
456
457     return (IPADM_SUCCESS);
458 }
459
460 /* ARGSUSED */
461 static ipadm_status_t
462 i_ipadm_set_usersrc(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;
469
470     /* if we are resetting, set the value to its default value */
471     if (flags & IPADM_OPT_DEFAULT)
472         pval = IPADM_NONESTR;
473
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) strncpy(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_privport(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     }
747     return (status);
748 }

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];

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     }

770     return (i_ipadm_set_prop(iph, arg, pdp, pval, proto, flags));
771 }

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;

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);

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;

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 }

829     return (status);
830 }

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;

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);

861         (void) snprintf(buf, *bufsize,
862 (val == 1 ? IPADM_ONSTR : IPADM_OFFSTR));
863     }
864 }

866 ret:
867     return (status);
868 }

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) strncpy(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) strncpy(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) strncpy(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) strncpy(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);
1029
1030         val = IPADM_OFFSTR;
1031         if (strcmp(pdp->ipd_name, "exchange_routes") == 0) {
1032             if (!(intf_flags & IFF_NOEXCH))
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 }
1054
1055     return (status);
1056 }
1057
1058 static void
1059 i_ipadm_perm2str(char *buf, uint_t *bufsize)
1060 {
1061     uint_t perm = atoi(buf);
1062
1063     (void) snprintf(buf, *bufsize, "%c%c",
1064         ((perm & MOD_PROP_PERM_READ) != 0) ? 'r' : '-',
1065         ((perm & MOD_PROP_PERM_WRITE) != 0) ? 'w' : '-');
1066 }
1067
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;
1079
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);
1084
1085     mip->mpr_version = MOD_PROP_VERSION;
1086     mip->mpr_flags = valtype;
1087     mip->mpr_proto = proto;
1088     if (ifname != NULL) {
1089         (void) strncpy(mip->mpr_ifname, ifname,
1090             sizeof (mip->mpr_ifname));
1091     }
1092     (void) strncpy(mip->mpr_name, pname, sizeof (mip->mpr_name));
1093     mip->mpr_valsize = *bufsize;
1094
1095     if (i_ipadm_striocctl(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     }
1104
1105     free(mip);
1106     return (status);
1107 }
1108
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_addr_obj_s *ipaddr;
1123     uint_t class = pdp->ipd_class;
1124     uint_t proto = pdp->ipd_proto;
1125
1126     (void) strncpy(pargp->ia_pname, pdp->ipd_name,
1127         sizeof (pargp->ia_pname));
1128     if (pval != NULL)
1129         (void) strncpy(pargp->ia_pval, pval, sizeof (pargp->ia_pval));
1130
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) strncpy(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) strncpy(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) strcpy(pargp->ia_ifname, object,
1152             sizeof (pargp->ia_ifname));
1153         (void) strncpy(pargp->ia_module, ipadm_proto2str(proto),
1154             sizeof (pargp->ia_module));
1155         break;
1156     case IPADMPROP_CLASS_ADDR:
1157         ipaddr = object;
1158         (void) strcpy(pargp->ia_ifname, ipaddr->ipadm_ifname,
1159             sizeof (pargp->ia_ifname));
1160         (void) strcpy(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) strncpy(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) strncpy(mip->mpr_ifname, ifname,
1350             sizeof(mip->mpr_ifname));
1351     }

1353     (void) strncpy(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_striocctl(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 && strlen(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) strncpy(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).

```

```

1537 *
1538 * Further if proto == MOD_PROTO_NONE, then it walks through all the supported
1539 * protocol property tables.
1540 */
1541 ipadm_status_t
1542 ipadm_walk_proptbl(uint_t proto, uint_t class, ipadm_prop_wfunc_t *func,
1543 void *arg)
1544 {
1545     ipadm_prop_desc_t    *pdtbl;
1546     ipadm_status_t      status = IPADM_SUCCESS;
1547     int                  i;
1548     int                  count = A_CNT(protocols);
1549
1550     if (func == NULL)
1551         return (IPADM_INVALID_ARG);
1552
1553     switch (class) {
1554     case IPADMPROP_CLASS_ADDR:
1555         pdtbl = ipadm_addrprop_table;
1556         break;
1557     case IPADMPROP_CLASS_IF:
1558     case IPADMPROP_CLASS_MODULE:
1559         pdtbl = i_ipadm_get_propdesc_table(proto);
1560         if (pdtbl == NULL && proto != MOD_PROTO_NONE)
1561             return (IPADM_INVALID_ARG);
1562         break;
1563     default:
1564         return (IPADM_INVALID_ARG);
1565     }
1566
1567     if (pdtbl != NULL) {
1568         /*
1569          * proto will be MOD_PROTO_NONE in the case of
1570          * IPADMPROP_CLASS_ADDR.
1571          */
1572         i_ipadm_walk_proptbl(pdtbl, proto, class, func, arg);
1573     } else {
1574         /* Walk thru all the protocol tables, we support */
1575         for (i = 0; i < count; i++) {
1576             pdtbl = i_ipadm_get_propdesc_table(protocols[i]);
1577             i_ipadm_walk_proptbl(pdtbl, protocols[i], class, func,
1578                 arg);
1579         }
1580     }
1581     return (status);
1582 }
1583
1584 /*
1585 * Given a property name, walks through all the instances of a property name.
1586 * Some properties have two instances one for v4 interfaces and another for v6
1587 * interfaces. For example: MTU. MTU can have different values for v4 and v6.
1588 * Therefore there are two properties for 'MTU'.
1589 * This function invokes 'func' for every instance of property 'pname'
1590 */
1591 ipadm_status_t
1592 ipadm_walk_prop(const char *pname, uint_t proto, uint_t class,
1593 ipadm_prop_wfunc_t *func, void *arg)
1594 {
1595     ipadm_prop_desc_t    *pdtbl, *pdp;
1596     ipadm_status_t      status = IPADM_SUCCESS;
1597     boolean_t           matched = B_FALSE;
1598
1599     if (pname == NULL || func == NULL)
1600         return (IPADM_INVALID_ARG);

```

```

1603     switch (class) {
1604     case IPADMPROP_CLASS_ADDR:
1605         pdtbl = ipadm_addrprop_table;
1606         break;
1607     case IPADMPROP_CLASS_IF:
1608     case IPADMPROP_CLASS_MODULE:
1609         pdtbl = i_ipadm_get_propdesc_table(proto);
1610         break;
1611     default:
1612         return (IPADM_INVALID_ARG);
1613     }
1614
1615     if (pdtbl == NULL)
1616         return (IPADM_INVALID_ARG);
1617
1618     for (pdp = pdtbl; pdp->ipd_name != NULL; pdp++) {
1619         if (strcmp(pname, pdp->ipd_name) != 0)
1620             continue;
1621         if (!(pdp->ipd_proto & proto))
1622             continue;
1623         matched = B_TRUE;
1624         /* we found a match, call the callback function */
1625         if (func(arg, pdp->ipd_name, pdp->ipd_proto) == B_FALSE)
1626             break;
1627     }
1628     if (!matched)
1629         status = IPADM_PROP_UNKNOWN;
1630     return (status);
1631 }
1632
1633 /* ARGSUSED */
1634 ipadm_status_t
1635 i_ipadm_get_onoff(ipadm_handle_t iph, const void *arg, ipadm_prop_desc_t *dp,
1636 char *buf, uint_t *bufsize, uint_t proto, uint_t valtype)
1637 {
1638     (void) snprintf(buf, *bufsize, "%s,%s", IPADM_ONSTR, IPADM_OFFSTR);
1639     return (IPADM_SUCCESS);
1640 }
1641
1642 /*
1643 * Makes a door call to ipmgmt to retrieve the persisted property value
1644 */
1645 ipadm_status_t
1646 i_ipadm_get_persist_propval(ipadm_handle_t iph, ipadm_prop_desc_t *pdp,
1647 char *gbuf, uint_t *gbufsize, const void *object)
1648 {
1649     ipmgmt_prop_arg_t    parg;
1650     ipmgmt_getprop_rval_t rval, *rvalp;
1651     size_t                nbytes;
1652     int                    err = 0;
1653
1654     bzero(&parg, sizeof (parg));
1655     parg.ia_cmd = IPMGMT_CMD_GETPROP;
1656     i_ipadm_populate_proparg(&parg, pdp, NULL, object);
1657
1658     rvalp = &rval;
1659     err = ipadm_door_call(iph, &parg, sizeof (parg), (void **)&rvalp,
1660         sizeof (rval), B_FALSE);
1661     if (err == 0) {
1662         /* assert that rvalp was not reallocated */
1663         assert(rvalp == &rval);
1664
1665         /* 'ir_pval' contains the property value */
1666         nbytes = snprintf(gbuf, *gbufsize, "%s", rvalp->ir_pval);
1667         if (nbytes >= *gbufsize) {
1668             /* 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_omame2nname_map {
1779     char    *iom_omame;
1780     char    *iom_nname;
1781     uint_t  iom_proto;
1782 } ipadm_omame2nname_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_omame2nname_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;
1934
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);
1938
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) strncpy(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 }
```

new/usr/src/pkg/manifests/system-kernel.mf

1

```
*****
45632 Wed Jun 13 12:04:20 2012
new/usr/src/pkg/manifests/system-kernel.mf
%B
*****
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 #
21 #
22 #
23 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
24 #
25 #
26 #
27 # The default for payload-bearing actions in this package is to appear in the
28 # global zone only. See the include file for greater detail, as well as
29 # information about overriding the defaults.
30 #
31 <include global_zone_only_component>
32 <include system-kernel.man1m.inc>
33 <include system-kernel.man2.inc>
34 <include system-kernel.man4.inc>
35 <include system-kernel.man5.inc>
36 <include system-kernel.man7.inc>
37 <include system-kernel.man7d.inc>
38 <include system-kernel.man7fs.inc>
39 <include system-kernel.man7m.inc>
40 <include system-kernel.man7p.inc>
41 <include system-kernel.man9.inc>
42 <include system-kernel.man9e.inc>
43 <include system-kernel.man9f.inc>
44 <include system-kernel.man9p.inc>
45 <include system-kernel.man9s.inc>
46 set name=pkg.fmri value=pkg:/system/kernel@$(PKGVERS)
47 set name=pkg.description \
48     value="core kernel software for a specific instruction-set architecture"
49 set name=pkg.summary value="Core Solaris Kernel"
50 set name=info.classification value=org.opensolaris.category.2008:System/Core
51 set name=variant.arch value=$(ARCH)
52 dir path=boot group=sys
53 $(i386_ONLY)dir path=boot/acpi group=sys
54 $(i386_ONLY)dir path=boot/acpi/tables group=sys
55 dir path=boot/solaris group=sys
56 dir path=boot/solaris/bin group=sys
57 dir path=etc group=sys
58 dir path=etc/crypto group=sys
59 dir path=etc/sock2path.d group=sys
60 dir path=kernel group=sys
61 $(i386_ONLY)dir path=kernel/$(ARCH64) group=sys
```

new/usr/src/pkg/manifests/system-kernel.mf

2

```
62 dir path=kernel/crypto group=sys
63 dir path=kernel/crypto/$(ARCH64) group=sys
64 dir path=kernel/dacf group=sys
65 dir path=kernel/dacf/$(ARCH64) group=sys
66 dir path=kernel/drv group=sys
67 dir path=kernel/drv/$(ARCH64) group=sys
68 dir path=kernel/exec group=sys
69 dir path=kernel/exec/$(ARCH64) group=sys
70 dir path=kernel/fs group=sys
71 dir path=kernel/fs/$(ARCH64) group=sys
72 dir path=kernel/ipp group=sys
73 dir path=kernel/ipp/$(ARCH64) group=sys
74 dir path=kernel/kiconv group=sys
75 dir path=kernel/kiconv/$(ARCH64) group=sys
76 dir path=kernel/mac group=sys
77 dir path=kernel/mac/$(ARCH64) group=sys
78 dir path=kernel/misc group=sys
79 dir path=kernel/misc/$(ARCH64) group=sys
80 dir path=kernel/misc/scsi_vhci group=sys
81 dir path=kernel/misc/scsi_vhci/$(ARCH64) group=sys
82 dir path=kernel/sched group=sys
83 dir path=kernel/sched/$(ARCH64) group=sys
84 dir path=kernel/socketmod group=sys
85 dir path=kernel/socketmod/$(ARCH64) group=sys
86 dir path=kernel/strmod group=sys
87 dir path=kernel/strmod/$(ARCH64) group=sys
88 dir path=kernel/sys group=sys
89 dir path=kernel/sys/$(ARCH64) group=sys
90 dir path=lib
91 dir path=lib/svc
92 dir path=lib/svc/manifest group=sys
93 dir path=lib/svc/manifest/system group=sys
94 dir path=lib/svc/method
95 dir path=usr/share/man
96 dir path=usr/share/man/man1m
97 dir path=usr/share/man/man2
98 dir path=usr/share/man/man3
99 dir path=usr/share/man/man4
100 dir path=usr/share/man/man5
101 dir path=usr/share/man/man7d
102 dir path=usr/share/man/man7fs
103 dir path=usr/share/man/man7m
104 dir path=usr/share/man/man7p
105 dir path=usr/share/man/man9
106 dir path=usr/share/man/man9e
107 dir path=usr/share/man/man9f
108 dir path=usr/share/man/man9p
109 dir path=usr/share/man/man9s
110 $(i386_ONLY)driver name=acpi_drv perms="* 0666 root sys"
111 driver name=aggr perms="* 0666 root sys"
112 driver name=arp perms="arp 0666 root sys"
113 driver name=bl perms="* 0666 root sys"
114 driver name=bridge clone_perms="bridge 0666 root sys" \
115     policy="read_priv_set=net_rawaccess write_priv_set=net_rawaccess"
116 $(sparc_ONLY)driver name=bscibus alias=SUNW,bscibus
117 $(i386_ONLY)driver name=bscibus alias=SVI0101
118 $(sparc_ONLY)driver name=bsciv alias=SUNW,bsciv perms="* 0644 root sys"
119 $(i386_ONLY)driver name=bsciv
120 driver name=clone
121 driver name=cn perms="* 0620 root tty"
122 driver name=conskbd perms="kbd 0666 root sys"
123 driver name=consms perms="mouse 0666 root sys"
124 driver name=cpuid perms="self 0644 root sys"
125 $(i386_ONLY)driver name=cpunex alias=cpus
126 driver name=crypto perms="crypto 0666 root sys"
127 driver name=cryptoadm perms="cryptoadm 0644 root sys"
```

```

128 $(sparc_ONLY)driver name=dad alias=ide-disk perms="* 0640 root sys"
129 driver name=dccp perms="dccp 0666 root sys"
130 driver name=dccp6 perms="dccp6 0666 root sys"
131 #endif /* ! codereview */
132 driver name=devinfo perms="devinfo 0640 root sys" \
133 perms="devinfo,ro 0444 root sys"
134 driver name=dld perms="* 0666 root sys"
135 driver name=dlpistub perms="* 0666 root sys"
136 $(sparc_ONLY)driver name=i8042 alias=8042
137 $(i386_ONLY)driver name=i8042
138 driver name=icmp perms="icmp 0666 root sys" \
139 policy="read_priv_set=net_icmpaccess write_priv_set=net_icmpaccess"
140 driver name=icmp6 perms="icmp6 0666 root sys" \
141 policy="read_priv_set=net_icmpaccess write_priv_set=net_icmpaccess"
142 $(i386_ONLY)driver name=intel_nb5000 \
143 alias=pci8086,25c0 \
144 alias=pci8086,25d0 \
145 alias=pci8086,25d4 \
146 alias=pci8086,25d8 \
147 alias=pci8086,3600 \
148 alias=pci8086,4000 \
149 alias=pci8086,4001 \
150 alias=pci8086,4003 \
151 alias=pci8086,65c0
152 $(i386_ONLY)driver name=intel_nhm \
153 alias=pci8086,3423 \
154 alias=pci8086,372a
155 $(i386_ONLY)driver name=intel_nhmex alias=pci8086,3438
156 driver name=ip perms="ip 0666 root sys" \
157 policy="read_priv_set=net_rawaccess write_priv_set=net_rawaccess"
158 driver name=ip6 perms="ip6 0666 root sys" \
159 policy="read_priv_set=net_rawaccess write_priv_set=net_rawaccess"
160 driver name=ipnet perms="lo0 0666 root sys" \
161 policy="read_priv_set=net_observability write_priv_set=net_observability"
162 driver name=ippctl
163 driver name=ipsecah perms="ipsecah 0666 root sys" \
164 policy="read_priv_set=sys_ip_config write_priv_set=sys_ip_config"
165 driver name=ipsecesp perms="ipsecesp 0666 root sys" \
166 policy="read_priv_set=sys_ip_config write_priv_set=sys_ip_config"
167 driver name=iptun
168 driver name=iwscn
169 driver name=kb8042 alias=pnpPNP,303
170 driver name=keysock perms="keysock 0666 root sys" \
171 policy="read_priv_set=sys_ip_config write_priv_set=sys_ip_config"
172 driver name=kmdb
173 driver name=kssl perms="* 0666 root sys"
174 driver name=llcl clone_perms="llcl 0666 root sys"
175 driver name=lofi perms="* 0600 root sys" perms="ctl 0644 root sys"
176 driver name=log perms="conslog 0666 root sys" perms="log 0640 root sys"
177 $(i386_ONLY)driver name=mc-amd \
178 alias=pci1022,1100 \
179 alias=pci1022,1101 \
180 alias=pci1022,1102
181 driver name=mm perms="allkmem 0600 root sys" perms="kmem 0640 root sys" \
182 perms="mem 0640 root sys" perms="null 0666 root sys" \
183 perms="zero 0666 root sys" \
184 policy="allkmem read_priv_set=all write_priv_set=all" \
185 policy="kmem read_priv_set=none write_priv_set=all" \
186 policy="mem read_priv_set=none write_priv_set=all"
187 driver name=mouse8042 alias=pnpPNP,f03
188 $(i386_ONLY)driver name=mpt class=scsi \
189 alias=pci1000,30 \
190 alias=pci1000,50 \
191 alias=pci1000,54 \
192 alias=pci1000,56 \
193 alias=pci1000,58 \

```

```

194 alias=pci1000,62 \
195 alias=pciex1000,56 \
196 alias=pciex1000,58 \
197 alias=pciex1000,62
198 driver name=nulldriver \
199 alias=scsa,nodev \
200 alias=scsa,probe
201 driver name=openeepc perms="openprom 0640 root sys" policy=write_priv_set=all
202 driver name=options
203 $(sparc_ONLY)driver name=pci_pci class=pci \
204 alias=pci1011,1 \
205 alias=pci1011,21 \
206 alias=pci1011,24 \
207 alias=pci1011,25 \
208 alias=pci1011,26 \
209 alias=pci1014,22 \
210 alias=pciclass,060400
211 $(i386_ONLY)driver name=pci_pci class=pci \
212 alias=pci1011,1 \
213 alias=pci1011,21 \
214 alias=pci1014,22 \
215 alias=pciclass,060400 \
216 alias=pciclass,060401
217 $(sparc_ONLY)driver name=pcieb \
218 alias=pciex108e,9010 \
219 alias=pciex108e,9020 \
220 alias=pciex10b5,8114 \
221 alias=pciex10b5,8516 \
222 alias=pciex10b5,8517 \
223 alias=pciex10b5,8518 \
224 alias=pciex10b5,8532 \
225 alias=pciex10b5,8533 \
226 alias=pciex10b5,8548 \
227 alias=pciexclass,060400
228 $(i386_ONLY)driver name=pcieb \
229 alias=pciexclass,060400 \
230 alias=pciexclass,060401
231 $(sparc_ONLY)driver name=pcieb_bcm alias=pciex1166,103
232 driver name=phymem perms="* 0600 root sys"
233 driver name=poll perms="* 0666 root sys"
234 $(sparc_ONLY)driver name=power alias=ali1535d+-power
235 $(i386_ONLY)driver name=power
236 driver name=pseudo alias=zconsnex
237 driver name=ptc perms="* 0666 root sys"
238 driver name=ptsl perms="* 0666 root sys"
239 $(sparc_ONLY)driver name=ramdisk alias=SUNW,ramdisk perms="* 0600 root sys" \
240 perms="ctl 0644 root sys"
241 $(i386_ONLY)driver name=ramdisk perms="* 0600 root sys" \
242 perms="ctl 0644 root sys"
243 driver name=random perms="* 0644 root sys" policy=write_priv_set=sys_devices
244 driver name=rts perms="rts 0666 root sys"
245 driver name=sad perms="admin 0666 root sys" perms="user 0666 root sys"
246 driver name=scsi_vhci class=scsi-self-identifying perms="* 0666 root sys" \
247 policy="devctl write_priv_set=sys_devices"
248 $(sparc_ONLY)driver name=sd perms="* 0640 root sys" \
249 alias=ide-cdrom \
250 alias=scsiclass,00 \
251 alias=scsiclass,05
252 $(i386_ONLY)driver name=sd perms="* 0640 root sys" \
253 alias=scsiclass,00 \
254 alias=scsiclass,05
255 driver name=sgen perms="* 0600 root sys" \
256 alias=scsa,08.bfcp \
257 alias=scsa,08.bvhci
258 driver name=simnet clone_perms="simnet 0666 root sys" perms="* 0666 root sys"
259 $(i386_ONLY)driver name=smbios perms="smbios 0444 root sys"

```

```

260 driver name=softmac
261 driver name=spdssock perms="spdssock 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=vnuc clone_perms="vnuc 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)/bscibus 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)/consm 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)/iwsn 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)/openeep 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)/spsock 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/ipsec group=sys
479 file path=kernel/drv/ipsec.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/llcl group=sys
494 file path=kernel/drv/llcl.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/openepr group=sys
510 file path=kernel/drv/openepr.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/phymem group=sys
517 file path=kernel/drv/phymem.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

```

```

524 $(i386_ONLY)file path=kernel/drv/ptc group=sys
525 file path=kernel/drv/ptc.conf group=sys
526 $(i386_ONLY)file path=kernel/drv/ptsl group=sys
527 file path=kernel/drv/ptsl.conf group=sys
528 $(i386_ONLY)file path=kernel/drv/ramdisk group=sys
529 file path=kernel/drv/ramdisk.conf group=sys
530 $(i386_ONLY)file path=kernel/drv/random group=sys
531 file path=kernel/drv/random.conf group=sys
532 $(i386_ONLY)file path=kernel/drv/rts group=sys
533 file path=kernel/drv/rts.conf group=sys
534 $(i386_ONLY)file path=kernel/drv/sad group=sys
535 file path=kernel/drv/sad.conf group=sys
536 $(i386_ONLY)file path=kernel/drv/scsi_vhci group=sys
537 file path=kernel/drv/scsi_vhci.conf group=sys \
538     original_name=SUNWckr:kernel/drv/scsi_vhci.conf preserve=true
539 $(sparc_ONLY)file path=kernel/drv/sd.conf group=sys \
540     original_name=SUNWckr:kernel/drv/sd.conf preserve=true
541 $(i386_ONLY)file path=kernel/drv/sgen group=sys
542 file path=kernel/drv/sgen.conf group=sys \
543     original_name=SUNWckr:kernel/drv/sgen.conf preserve=true
544 $(i386_ONLY)file path=kernel/drv/simnet group=sys
545 file path=kernel/drv/simnet.conf group=sys
546 $(i386_ONLY)file path=kernel/drv/smbios group=sys
547 $(i386_ONLY)file path=kernel/drv/smbios.conf group=sys
548 $(i386_ONLY)file path=kernel/drv/softmac group=sys
549 file path=kernel/drv/softmac.conf group=sys
550 $(i386_ONLY)file path=kernel/drv/spdsock group=sys
551 file path=kernel/drv/spdsock.conf group=sys
552 $(i386_ONLY)file path=kernel/drv/st group=sys
553 file path=kernel/drv/st.conf group=sys \
554     original_name=SUNWckr:kernel/drv/st.conf preserve=true
555 $(i386_ONLY)file path=kernel/drv/sy group=sys
556 file path=kernel/drv/sy.conf group=sys
557 $(i386_ONLY)file path=kernel/drv/sysevent group=sys
558 file path=kernel/drv/sysevent.conf group=sys
559 $(i386_ONLY)file path=kernel/drv/sysmsg group=sys
560 file path=kernel/drv/sysmsg.conf group=sys
561 $(i386_ONLY)file path=kernel/drv/tcp group=sys
562 file path=kernel/drv/tcp.conf group=sys
563 $(i386_ONLY)file path=kernel/drv/tcp6 group=sys
564 file path=kernel/drv/tcp6.conf group=sys
565 $(i386_ONLY)file path=kernel/drv/tl group=sys
566 file path=kernel/drv/tl.conf group=sys
567 $(i386_ONLY)file path=kernel/drv/tzmon group=sys
568 $(i386_ONLY)file path=kernel/drv/tzmon.conf group=sys
569 $(sparc_ONLY)file path=kernel/drv/uata.conf group=sys \
570     original_name=SUNWckr:kernel/drv/uata.conf preserve=true
571 $(i386_ONLY)file path=kernel/drv/ucode group=sys
572 $(i386_ONLY)file path=kernel/drv/ucode.conf group=sys
573 $(i386_ONLY)file path=kernel/drv/udp group=sys
574 file path=kernel/drv/udp.conf group=sys
575 $(i386_ONLY)file path=kernel/drv/udp6 group=sys
576 file path=kernel/drv/udp6.conf group=sys
577 $(i386_ONLY)file path=kernel/drv/vgatext group=sys
578 $(i386_ONLY)file path=kernel/drv/vnic group=sys
579 file path=kernel/drv/vnic.conf group=sys
580 $(i386_ONLY)file path=kernel/drv/wc group=sys
581 file path=kernel/drv/wc.conf group=sys
582 $(sparc_ONLY)file path=kernel/exec/$(ARCH64)/aoutexec group=sys mode=0755
583 file path=kernel/exec/$(ARCH64)/elfexec group=sys mode=0755
584 file path=kernel/exec/$(ARCH64)/intpexec group=sys mode=0755
585 $(i386_ONLY)file path=kernel/exec/elfexec group=sys mode=0755
586 $(i386_ONLY)file path=kernel/exec/intpexec group=sys mode=0755
587 file path=kernel/fs/$(ARCH64)/autofs group=sys mode=0755
588 file path=kernel/fs/$(ARCH64)/cachefs group=sys mode=0755
589 file path=kernel/fs/$(ARCH64)/ctfs group=sys mode=0755

```

```

590 file path=kernel/fs/$(ARCH64)/dcfs group=sys mode=0755
591 file path=kernel/fs/$(ARCH64)/dev group=sys mode=0755
592 file path=kernel/fs/$(ARCH64)/devfs group=sys mode=0755
593 file path=kernel/fs/$(ARCH64)/fifofs group=sys mode=0755
594 file path=kernel/fs/$(ARCH64)/hsfs group=sys mode=0755
595 file path=kernel/fs/$(ARCH64)/lofs group=sys mode=0755
596 file path=kernel/fs/$(ARCH64)/mntfs group=sys mode=0755
597 file path=kernel/fs/$(ARCH64)/namefs group=sys mode=0755
598 file path=kernel/fs/$(ARCH64)/objfs group=sys mode=0755
599 file path=kernel/fs/$(ARCH64)/procfs group=sys mode=0755
600 file path=kernel/fs/$(ARCH64)/sharefs group=sys mode=0755
601 file path=kernel/fs/$(ARCH64)/sockfs group=sys mode=0755
602 file path=kernel/fs/$(ARCH64)/specfs group=sys mode=0755
603 file path=kernel/fs/$(ARCH64)/tmpfs group=sys mode=0755
604 file path=kernel/fs/$(ARCH64)/ufs group=sys mode=0755
605 $(i386_ONLY)file path=kernel/fs/autofs group=sys mode=0755
606 $(i386_ONLY)file path=kernel/fs/cachefs group=sys mode=0755
607 $(i386_ONLY)file path=kernel/fs/ctfs group=sys mode=0755
608 $(i386_ONLY)file path=kernel/fs/dcfs group=sys mode=0755
609 $(i386_ONLY)file path=kernel/fs/dev group=sys mode=0755
610 $(i386_ONLY)file path=kernel/fs/devfs group=sys mode=0755
611 $(i386_ONLY)file path=kernel/fs/fifofs group=sys mode=0755
612 $(i386_ONLY)file path=kernel/fs/hsfs group=sys mode=0755
613 $(i386_ONLY)file path=kernel/fs/lofs group=sys mode=0755
614 $(i386_ONLY)file path=kernel/fs/mntfs group=sys mode=0755
615 $(i386_ONLY)file path=kernel/fs/namefs group=sys mode=0755
616 $(i386_ONLY)file path=kernel/fs/objfs group=sys mode=0755
617 $(i386_ONLY)file path=kernel/fs/procfs group=sys mode=0755
618 $(i386_ONLY)file path=kernel/fs/sharefs group=sys mode=0755
619 $(i386_ONLY)file path=kernel/fs/sockfs group=sys mode=0755
620 $(i386_ONLY)file path=kernel/fs/specfs group=sys mode=0755
621 $(i386_ONLY)file path=kernel/fs/tmpfs group=sys mode=0755
622 $(i386_ONLY)file path=kernel/fs/ufs group=sys mode=0755
623 $(i386_ONLY)file path=kernel/genunix group=sys mode=0755
624 file path=kernel/ipp/$(ARCH64)/ippgc group=sys mode=0755
625 $(i386_ONLY)file path=kernel/ipp/ippgc group=sys mode=0755
626 file path=kernel/kiconv/$(ARCH64)/kiconv_emea group=sys mode=0755
627 file path=kernel/kiconv/$(ARCH64)/kiconv_ja group=sys mode=0755
628 file path=kernel/kiconv/$(ARCH64)/kiconv_ko group=sys mode=0755
629 file path=kernel/kiconv/$(ARCH64)/kiconv_sc group=sys mode=0755
630 file path=kernel/kiconv/$(ARCH64)/kiconv_tc group=sys mode=0755
631 $(i386_ONLY)file path=kernel/kiconv/kiconv_emea group=sys mode=0755
632 $(i386_ONLY)file path=kernel/kiconv/kiconv_ja group=sys mode=0755
633 $(i386_ONLY)file path=kernel/kiconv/kiconv_ko group=sys mode=0755
634 $(i386_ONLY)file path=kernel/kiconv/kiconv_sc group=sys mode=0755
635 $(i386_ONLY)file path=kernel/kiconv/kiconv_tc group=sys mode=0755
636 file path=kernel/mac/$(ARCH64)/mac_6to4 group=sys mode=0755
637 file path=kernel/mac/$(ARCH64)/mac_ether group=sys mode=0755
638 file path=kernel/mac/$(ARCH64)/mac_ib group=sys mode=0755
639 file path=kernel/mac/$(ARCH64)/mac_ipv4 group=sys mode=0755
640 file path=kernel/mac/$(ARCH64)/mac_ipv6 group=sys mode=0755
641 file path=kernel/mac/$(ARCH64)/mac_wifi group=sys mode=0755
642 $(i386_ONLY)file path=kernel/mac/mac_6to4 group=sys mode=0755
643 $(i386_ONLY)file path=kernel/mac/mac_ether group=sys mode=0755
644 $(i386_ONLY)file path=kernel/mac/mac_ib group=sys mode=0755
645 $(i386_ONLY)file path=kernel/mac/mac_ipv4 group=sys mode=0755
646 $(i386_ONLY)file path=kernel/mac/mac_ipv6 group=sys mode=0755
647 $(i386_ONLY)file path=kernel/mac/mac_wifi group=sys mode=0755
648 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/acpica group=sys mode=0755
649 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/agpmaster group=sys mode=0755
650 file path=kernel/misc/$(ARCH64)/bignum group=sys mode=0755
651 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/bootdev group=sys mode=0755
652 file path=kernel/misc/$(ARCH64)/busra group=sys mode=0755
653 file path=kernel/misc/$(ARCH64)/cardbus group=sys mode=0755
654 file path=kernel/misc/$(ARCH64)/cmlb group=sys mode=0755
655 file path=kernel/misc/$(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)/iomulib 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)/swappgeneric 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/iomulib 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

```

```

788 $(i386_ONLY)file path=kernel/strmod/pckt group=sys mode=0755
789 $(i386_ONLY)file path=kernel/strmod/pfmod group=sys mode=0755
790 $(i386_ONLY)file path=kernel/strmod/pipemod group=sys mode=0755
791 $(i386_ONLY)file path=kernel/strmod/pitem group=sys mode=0755
792 $(i386_ONLY)file path=kernel/strmod/redirmod group=sys mode=0755
793 $(i386_ONLY)file path=kernel/strmod/rpcmod group=sys mode=0755
794 $(i386_ONLY)file path=kernel/strmod/timod group=sys mode=0755
795 $(i386_ONLY)file path=kernel/strmod/tirdwr group=sys mode=0755
796 $(i386_ONLY)file path=kernel/strmod/ttcompat group=sys mode=0755
797 file path=kernel/sys/$(ARCH64)/c2audit group=sys mode=0755
798 file path=kernel/sys/$(ARCH64)/doorfs group=sys mode=0755
799 file path=kernel/sys/$(ARCH64)/inst_sync group=sys mode=0755
800 file path=kernel/sys/$(ARCH64)/kaio group=sys mode=0755
801 file path=kernel/sys/$(ARCH64)/msgsys group=sys mode=0755
802 file path=kernel/sys/$(ARCH64)/pipe group=sys mode=0755
803 file path=kernel/sys/$(ARCH64)/portfs group=sys mode=0755
804 file path=kernel/sys/$(ARCH64)/pset group=sys mode=0755
805 file path=kernel/sys/$(ARCH64)/semsys group=sys mode=0755
806 file path=kernel/sys/$(ARCH64)/shmsys group=sys mode=0755
807 $(i386_ONLY)file path=kernel/sys/c2audit group=sys mode=0755
808 $(i386_ONLY)file path=kernel/sys/doorfs group=sys mode=0755
809 $(i386_ONLY)file path=kernel/sys/inst_sync group=sys mode=0755
810 $(i386_ONLY)file path=kernel/sys/kaio group=sys mode=0755
811 $(i386_ONLY)file path=kernel/sys/msgsys group=sys mode=0755
812 $(i386_ONLY)file path=kernel/sys/pipe group=sys mode=0755
813 $(i386_ONLY)file path=kernel/sys/portfs group=sys mode=0755
814 $(i386_ONLY)file path=kernel/sys/pset group=sys mode=0755
815 $(i386_ONLY)file path=kernel/sys/semsys group=sys mode=0755
816 $(i386_ONLY)file path=kernel/sys/shmsys group=sys mode=0755
817 file path=lib/svc/manifest/system/dumpadm.xml group=sys mode=0444
818 file path=lib/svc/manifest/system/intrd.xml group=sys mode=0444
819 file path=lib/svc/manifest/system/scheduler.xml group=sys mode=0444
820 file path=lib/svc/method/svc-dumpadm mode=0555
821 file path=lib/svc/method/svc-intrd mode=0555
822 file path=lib/svc/method/svc-scheduler mode=0555
823 $(sparc_ONLY)file path=usr/share/man/man1m/monitor.1m
824 $(sparc_ONLY)file path=usr/share/man/man1m/obpsym.1m
825 # On SPARC driver/bscv is Serverbladel specific, and in system/kernel/platform
826 # We keep the manual page generic
827 $(sparc_ONLY)file path=usr/share/man/man7d/dad.7d
828 $(i386_ONLY)file path=usr/share/man/man7d/smbios.7d
829 # Sadly vuid mouse support is in different packages on different platforms
830 # While kstat(7D) is in SUNWcs, the structures are general
831 hardlink path=kernel/misc/$(ARCH64)/md5 \
832 target=../../../../kernel/crypto/$(ARCH64)/md5
833 hardlink path=kernel/misc/$(ARCH64)/sha1 \
834 target=../../../../kernel/crypto/$(ARCH64)/sha1
835 hardlink path=kernel/misc/$(ARCH64)/sha2 \
836 target=../../../../kernel/crypto/$(ARCH64)/sha2
837 $(i386_ONLY)hardlink path=kernel/misc/md5 target=../../../../kernel/crypto/md5
838 $(i386_ONLY)hardlink path=kernel/misc/sha1 target=../../../../kernel/crypto/sha1
839 $(i386_ONLY)hardlink path=kernel/misc/sha2 target=../../../../kernel/crypto/sha2
840 hardlink path=kernel/socketmod/$(ARCH64)/dccp \
841 target=../../../../kernel/drv/$(ARCH64)/dccp
842 #endif /* ! codereview */
843 hardlink path=kernel/socketmod/$(ARCH64)/icmp \
844 target=../../../../kernel/drv/$(ARCH64)/icmp
845 hardlink path=kernel/socketmod/$(ARCH64)/rts \
846 target=../../../../kernel/drv/$(ARCH64)/rts
847 hardlink path=kernel/socketmod/$(ARCH64)/tcp \
848 target=../../../../kernel/drv/$(ARCH64)/tcp
849 hardlink path=kernel/socketmod/$(ARCH64)/udp \
850 target=../../../../kernel/drv/$(ARCH64)/udp
851 $(i386_ONLY)hardlink path=kernel/socketmod/dccp target=../../../../kernel/drv/dccp
852 #endif /* ! codereview */
853 $(i386_ONLY)hardlink path=kernel/socketmod/icmp target=../../../../kernel/drv/icmp

```

```

854 $(i386_ONLY)hardlink path=kernel/socketmod/rts target=../../../../kernel/drv/rts
855 $(i386_ONLY)hardlink path=kernel/socketmod/tcp target=../../../../kernel/drv/tcp
856 $(i386_ONLY)hardlink path=kernel/socketmod/udp target=../../../../kernel/drv/udp
857 hardlink path=kernel/strmod/$(ARCH64)/arp \
858 target=../../../../kernel/drv/$(ARCH64)/arp
859 hardlink path=kernel/strmod/$(ARCH64)/dccp \
860 target=../../../../kernel/drv/$(ARCH64)/dccp
861 #endif /* ! codereview */
862 hardlink path=kernel/strmod/$(ARCH64)/icmp \
863 target=../../../../kernel/drv/$(ARCH64)/icmp
864 hardlink path=kernel/strmod/$(ARCH64)/ip \
865 target=../../../../kernel/drv/$(ARCH64)/ip
866 hardlink path=kernel/strmod/$(ARCH64)/ipsec \
867 target=../../../../kernel/drv/$(ARCH64)/ipsec
868 hardlink path=kernel/strmod/$(ARCH64)/ipsecep \
869 target=../../../../kernel/drv/$(ARCH64)/ipsecep
870 hardlink path=kernel/strmod/$(ARCH64)/keysock \
871 target=../../../../kernel/drv/$(ARCH64)/keysock
872 hardlink path=kernel/strmod/$(ARCH64)/tcp \
873 target=../../../../kernel/drv/$(ARCH64)/tcp
874 hardlink path=kernel/strmod/$(ARCH64)/udp \
875 target=../../../../kernel/drv/$(ARCH64)/udp
876 $(i386_ONLY)hardlink path=kernel/strmod/arp target=../../../../kernel/drv/arp
877 $(i386_ONLY)hardlink path=kernel/strmod/dccp target=../../../../kernel/drv/dccp
878 #endif /* ! codereview */
879 $(i386_ONLY)hardlink path=kernel/strmod/icmp target=../../../../kernel/drv/icmp
880 $(i386_ONLY)hardlink path=kernel/strmod/ip target=../../../../kernel/drv/ip
881 $(i386_ONLY)hardlink path=kernel/strmod/ipsec \
882 target=../../../../kernel/drv/ipsec
883 $(i386_ONLY)hardlink path=kernel/strmod/ipsecep \
884 target=../../../../kernel/drv/ipsecep
885 $(i386_ONLY)hardlink path=kernel/strmod/keysock \
886 target=../../../../kernel/drv/keysock
887 $(i386_ONLY)hardlink path=kernel/strmod/tcp target=../../../../kernel/drv/tcp
888 $(i386_ONLY)hardlink path=kernel/strmod/udp target=../../../../kernel/drv/udp
889 hardlink path=kernel/sys/$(ARCH64)/autofs \
890 target=../../../../kernel/fs/$(ARCH64)/autofs
891 hardlink path=kernel/sys/$(ARCH64)/rpcmod \
892 target=../../../../kernel/strmod/$(ARCH64)/rpcmod
893 $(i386_ONLY)hardlink path=kernel/sys/autofs target=../../../../kernel/fs/autofs
894 $(i386_ONLY)hardlink path=kernel/sys/rpcmod target=../../../../kernel/strmod/rpcmod
895 legacy pkg=SUNWckr \
896 desc="core kernel software for a specific instruction-set architecture" \
897 name="Core Solaris Kernel (Root)"
898 license cr_Sun license=cr_Sun
899 license lic_CDDL license=lic_CDDL
900 license usr/src/cmd/adb/common/libstand/THIRDPARTYLICENSE \
901 license usr/src/cmd/adb/common/libstand/THIRDPARTYLICENSE
902 license usr/src/common/bzip2/LICENSE license=usr/src/common/bzip2/LICENSE
903 license usr/src/common/crypto/THIRDPARTYLICENSE.cryptogams \
904 license=usr/src/common/crypto/THIRDPARTYLICENSE.cryptogams
905 $(i386_ONLY)license usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.gladman \
906 license=usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.gladman
907 $(i386_ONLY)license usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.openssl \
908 license=usr/src/common/crypto/aes/amd64/THIRDPARTYLICENSE.openssl
909 license usr/src/common/crypto/ecc/THIRDPARTYLICENSE \
910 license=usr/src/common/crypto/ecc/THIRDPARTYLICENSE
911 $(i386_ONLY)license usr/src/common/crypto/md5/amd64/THIRDPARTYLICENSE \
912 license=usr/src/common/crypto/md5/amd64/THIRDPARTYLICENSE
913 license usr/src/common/mpi/THIRDPARTYLICENSE \
914 license=usr/src/common/mpi/THIRDPARTYLICENSE
915 license usr/src/uts/common/inet/ip/THIRDPARTYLICENSE.rts \
916 license=usr/src/uts/common/inet/ip/THIRDPARTYLICENSE.rts
917 license usr/src/uts/common/inet/tcp/THIRDPARTYLICENSE \
918 license=usr/src/uts/common/inet/tcp/THIRDPARTYLICENSE
919 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
```

```

*****
42921 Wed Jun 13 12:04:22 2012
new/usr/src/uts/common/Makefile.files
%B
*****
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 #
21 #
22 #
23 # Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
24 #
25 #
26 #
27 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
28 #
29 #
30 #
31 # This Makefile defines all file modules for the directory uts/common
32 # and its children. These are the source files which may be considered
33 # common to all SunOS systems.
34 #
35 i386_CORE_OBJS += \
36     atomic.o      \
37     avintr.o      \
38     pic.o
39 #
40 sparc_CORE_OBJS +=
41 #
42 COMMON_CORE_OBJS +=
43     beep.o        \
44     bitset.o      \
45     bp_map.o      \
46     brand.o       \
47     cpucaps.o     \
48     cmt.o         \
49     cmt_policy.o  \
50     cpu.o         \
51     cpu_event.o   \
52     cpu_intr.o    \
53     cpu_pm.o      \
54     cpupart.o     \
55     cap_util.o    \
56     disp.o        \
57     group.o       \
58     kstat_fr.o    \
59     iscsiboot_prop.o \
60     lgrp.o        \
61     lgrp_topo.o   \

```

```

62     mmapobj.o     \
63     mutex.o       \
64     page_lock.o   \
65     page_retire.o \
66     panic.o       \
67     param.o       \
68     pg.o          \
69     pghw.o        \
70     putnext.o     \
71     rctl_proc.o   \
72     rwlock.o      \
73     seg_kmem.o    \
74     softint.o     \
75     string.o      \
76     strtol.o      \
77     strtoul.o     \
78     strtoll.o     \
79     strtoull.o    \
80     thread_intr.o \
81     vm_page.o     \
82     vm_pagelist.o \
83     zlib_obj.o    \
84     clock_tick.o

86 CORE_OBJS += $(COMMON_CORE_OBJS) $(MACH)_CORE_OBJS

88 ZLIB_OBJS =
89     zutil.o zmod.o zmod_subr.o \
90     Adler32.o crc32.o deflate.o inffast.o \
91     inflate.o inftrees.o trees.o

92 GENUNIX_OBJS += \
93     access.o \
94     acl.o \
95     acl_common.o \
96     adjtime.o \
97     alarm.o \
98     aio_subr.o \
99     auditsys.o \
100    audit_core.o \
101    audit_zone.o \
102    audit_memory.o \
103    autoconf.o \
104    avl.o \
105    bdev_dsort.o \
106    bio.o \
107    bitmap.o \
108    blabel.o \
109    brandsys.o \
110    bz2blocksort.o \
111    bz2compress.o \
112    bz2decompress.o \
113    bz2randtable.o \
114    bz2bzlib.o \
115    bz2crctable.o \
116    bz2huffman.o \
117    callb.o \
118    callout.o \
119    chdir.o \
120    chmod.o \
121    chown.o \
122    cladm.o \
123    class.o \
124    clock.o \
125    clock_highres.o \
126    clock_realtime.o \
127    close.o \

```

new/usr/src/uts/common/Makefile.files

```

128 compress.o \
129 condvar.o \
130 conf.o \
131 console.o \
132 contract.o \
133 copyops.o \
134 core.o \
135 corectl.o \
136 cred.o \
137 cs_stubs.o \
138 dacf.o \
139 dacf_clnt.o \
140 damap.o \
141 cyclic.o \
142 ddi.o \
143 ddifm.o \
144 ddi_hp_impl.o \
145 ddi_hp_ndi.o \
146 ddi_intr.o \
147 ddi_intr_impl.o \
148 ddi_intr_irm.o \
149 ddi_nodeid.o \
150 ddi_timer.o \
151 devcfg.o \
152 devcache.o \
153 device.o \
154 devid.o \
155 devid_cache.o \
156 devid_scsi.o \
157 devid_smp.o \
158 devpolicy.o \
159 disp_lock.o \
160 dnlc.o \
161 driver.o \
162 dumpsubr.o \
163 driver_lyr.o \
164 dtrace_subr.o \
165 errorq.o \
166 etheraddr.o \
167 evchannels.o \
168 exacct.o \
169 exacct_core.o \
170 exec.o \
171 exit.o \
172 fbio.o \
173 fcntl.o \
174 fdbuffer.o \
175 fdsync.o \
176 fem.o \
177 ffs.o \
178 fio.o \
179 flock.o \
180 fm.o \
181 fork.o \
182 vpm.o \
183 fs_reparse.o \
184 fs_subr.o \
185 fsflush.o \
186 ftrace.o \
187 getcwd.o \
188 getdents.o \
189 getloadavg.o \
190 getpagesizes.o \
191 getpid.o \
192 gfs.o \
193 rusagesys.o \

```

3

new/usr/src/uts/common/Makefile.files

```

194 gid.o \
195 groups.o \
196 grow.o \
197 hat_refmod.o \
198 id32.o \
199 id_space.o \
200 inet_ntop.o \
201 instance.o \
202 ioctl.o \
203 ip_cksum.o \
204 issetugid.o \
205 ippconf.o \
206 kopc.o \
207 kdi.o \
208 kiconv.o \
209 klpd.o \
210 kmem.o \
211 ksyms_snapshot.o \
212 l_strplumb.o \
213 labelsys.o \
214 link.o \
215 list.o \
216 lockstat_subr.o \
217 log_sysevent.o \
218 logsubr.o \
219 lookup.o \
220 lseek.o \
221 ltos.o \
222 lwp.o \
223 lwp_create.o \
224 lwp_info.o \
225 lwp_self.o \
226 lwp_sobj.o \
227 lwp_timer.o \
228 lwpsys.o \
229 main.o \
230 mmapobjs.o \
231 memcntl.o \
232 memstr.o \
233 lgrpsys.o \
234 mkdir.o \
235 mknod.o \
236 mount.o \
237 move.o \
238 msacct.o \
239 multidata.o \
240 nbmlock.o \
241 ndifm.o \
242 nice.o \
243 netstack.o \
244 ntptime.o \
245 nvpair.o \
246 nvpair_alloc_system.o \
247 nvpair_alloc_fixed.o \
248 octet.o \
249 open.o \
250 p_online.o \
251 pathconf.o \
252 pathname.o \
253 pause.o \
254 serializer.o \
255 pci_intr_lib.o \
256 pci_cap.o \
257 pcifm.o \
258 pgrp.o \
259 pgrpsys.o \

```

4

new/usr/src/uts/common/Makefile.files

```

260 pid.o \
261 pkp_hash.o \
262 policy.o \
263 poll.o \
264 pool.o \
265 pool_pset.o \
266 port_subr.o \
267 ppriv.o \
268 printf.o \
269 priocntl.o \
270 priv.o \
271 priv_const.o \
272 proc.o \
273 procset.o \
274 processor_bind.o \
275 processor_info.o \
276 profil.o \
277 project.o \
278 qsort.o \
279 rctl.o \
280 rctlsys.o \
281 readlink.o \
282 refstr.o \
283 rename.o \
284 resolvepath.o \
285 retire_store.o \
286 process.o \
287 rlimit.o \
288 rmap.o \
289 rw.o \
290 rwstlock.o \
291 sad_conf.o \
292 sid.o \
293 sidsys.o \
294 sched.o \
295 schedctl.o \
296 sctp_crc32.o \
297 seg_dev.o \
298 seg_kp.o \
299 seg_kpm.o \
300 seg_map.o \
301 seg_vn.o \
302 seg_spt.o \
303 semaphore.o \
304 sendfile.o \
305 session.o \
306 share.o \
307 shuttle.o \
308 sig.o \
309 sigaction.o \
310 sigaltstack.o \
311 signotify.o \
312 sigpending.o \
313 sigprocmask.o \
314 sigqueue.o \
315 sigsendset.o \
316 sigsuspend.o \
317 sigtimedwait.o \
318 sleepq.o \
319 sock_conf.o \
320 space.o \
321 sscanf.o \
322 stat.o \
323 statfs.o \
324 statvfs.o \
325 stol.o \

```

5

new/usr/src/uts/common/Makefile.files

```

326 str_conf.o \
327 strcalls.o \
328 stream.o \
329 streamio.o \
330 strext.o \
331 strsubr.o \
332 strsun.o \
333 subr.o \
334 sunddi.o \
335 sunmdi.o \
336 sunndi.o \
337 sunpci.o \
338 sunpm.o \
339 sundlpi.o \
340 suntpi.o \
341 swap_subr.o \
342 swap_vnops.o \
343 symlink.o \
344 sync.o \
345 sysclass.o \
346 sysconfig.o \
347 sysent.o \
348 sysfs.o \
349 systeminfo.o \
350 task.o \
351 taskq.o \
352 tasksys.o \
353 time.o \
354 timer.o \
355 times.o \
356 timers.o \
357 thread.o \
358 tlabel.o \
359 tnf_res.o \
360 turnstile.o \
361 tty_common.o \
362 u8_textprep.o \
363 uadmin.o \
364 uconv.o \
365 ucredsys.o \
366 uid.o \
367 umask.o \
368 umount.o \
369 uname.o \
370 unix_bb.o \
371 unlink.o \
372 urw.o \
373 utime.o \
374 utssys.o \
375 uucopy.o \
376 vfs.o \
377 vfs_conf.o \
378 vmem.o \
379 vm_anon.o \
380 vm_as.o \
381 vm_meter.o \
382 vm_pageout.o \
383 vm_pvn.o \
384 vm_rm.o \
385 vm_seg.o \
386 vm_subr.o \
387 vm_swap.o \
388 vm_usage.o \
389 vnode.o \
390 vuid_queue.o \
391 vuid_store.o \

```

6

new/usr/src/uts/common/Makefile.files

7

```

392          waitq.o \
393          watchpoint.o \
394          yield.o \
395          scsi_confdata.o \
396          xattr.o \
397          xattr_common.o \
398          xdr_mblk.o \
399          xdr_mem.o \
400          xdr.o \
401          xdr_array.o \
402          xdr_refer.o \
403          xhat.o \
404          zone.o

406 #
407 #       Stubs for the stand-alone linker/loader
408 #
409 sparc_GENSTUBS_OBJS = \
410         kobj_stubs.o

412 i386_GENSTUBS_OBJS =

414 COMMON_GENSTUBS_OBJS =

416 GENSTUBS_OBJS += $(COMMON_GENSTUBS_OBJS) $($ (MACH)_GENSTUBS_OBJS)

418 #
419 #       DTrace and DTrace Providers
420 #
421 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

423 SDT_OBJS += sdt_subr.o

425 PROFILE_OBJS += profile.o

427 SYSTRACE_OBJS += systrace.o

429 LOCKSTAT_OBJS += lockstat.o

431 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

433 DCPC_OBJS += dcpc.o

435 #
436 #       Driver (pseudo-driver) Modules
437 #
438 IPP_OBJS += ippctl.o

440 AUDIO_OBJS += audio_client.o audio_ddi.o audio_engine.o \
441         audio_fltdata.o audio_format.o audio_ctrl.o \
442         audio_grc3.o audio_output.o audio_input.o \
443         audio_oss.o audio_sun.o

445 AUDIOEMU10K_OBJS += audioemu10k.o

447 AUDIOENS_OBJS += audioens.o

449 AUDIOVIA823X_OBJS += audiovia823x.o

451 AUDIOVIA97_OBJS += audiovia97.o

453 AUDIO1575_OBJS += audio1575.o

455 AUDIO810_OBJS += audio810.o

457 AUDIOCMI_OBJS += audiocmi.o

```

new/usr/src/uts/common/Makefile.files

8

```

459 AUDIOCMIHD_OBJS += audiocmihd.o

461 AUDIOHD_OBJS += audiohd.o

463 AUDIOIXP_OBJS += audioixp.o

465 AUDIOLS_OBJS += audiols.o

467 AUDIOP16X_OBJS += audiop16x.o

469 AUDIOPCI_OBJS += audiopci.o

471 AUDIOSOLO_OBJS += audiosolo.o

473 AUDIOTS_OBJS += audiots.o

475 AC97_OBJS += ac97.o ac97_ad.o ac97_alc.o ac97_cmi.o

477 BLKDEV_OBJS += blkdev.o

479 CARDBUS_OBJS += cardbus.o cardbus_hp.o cardbus_cfg.o

481 CONSKBD_OBJS += conskbd.o

483 CONSMS_OBJS += consms.o

485 OLDPTY_OBJS += tty_ptyconf.o

487 PTC_OBJS += tty_pty.o

489 PTSL_OBJS += tty_pts.o

491 PTM_OBJS += ptm.o

493 MII_OBJS += mii.o mii_cicada.o mii_natsemi.o mii_intel.o mii_qualsemi.o \
494         mii_marvell.o mii_realtek.o mii_other.o

496 PTS_OBJS += pts.o

498 PTY_OBJS += ptms_conf.o

500 SAD_OBJS += sad.o

502 MD4_OBJS += md4.o md4_mod.o

504 MD5_OBJS += md5.o md5_mod.o

506 SHA1_OBJS += sha1.o sha1_mod.o

508 SHA2_OBJS += sha2.o sha2_mod.o

510 IPGPC_OBJS += classifierddi.o classifier.o filters.o trie.o table.o \
511         ba_table.o

513 DSCPMK_OBJS += dscpmk.o dscpmkddi.o

515 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o

517 FLOWACCT_OBJS += flowacctddi.o flowacct.o

519 TOKENMT_OBJS += tokenmt.o tokenmtddi.o

521 TSWTCL_OBJS += tswtcl.o tswtclddi.o

523 ARP_OBJS += arpddi.o

```

```

525 ICMP_OBJS += icmpddi.o
527 ICMP6_OBJS += icmp6ddi.o
529 RTS_OBJS += rtsddi.o

531 IP_ICMP_OBJS = icmp.o icmp_opt_data.o
532 IP_RTS_OBJS = rts.o rts_opt_data.o
533 IP_TCP_OBJS = tcp.o tcp_fusion.o tcp_opt_data.o tcp_sack.o tcp_stats.o \
534 tcp_misc.o tcp_timers.o tcp_time_wait.o tcp_tpi.o tcp_output.o \
535 tcp_input.o tcp_socket.o tcp_bind.o tcp_cluster.o tcp_tunables.o
536 IP_UDP_OBJS = udp.o udp_opt_data.o udp_tunables.o udp_stats.o
537 IP_SCTP_OBJS = sctp.o sctp_opt_data.o sctp_output.o \
538 sctp_init.o sctp_input.o sctp_cookie.o \
539 sctp_conn.o sctp_error.o sctp_snmp.o \
540 sctp_tunables.o sctp_shutdown.o sctp_common.o \
541 sctp_timer.o sctp_heartbeat.o sctp_hash.o \
542 sctp_bind.o sctp_notify.o sctp_asconf.o \
543 sctp_addr.o tn_ipopt.o tnet.o ip_netinfo.o \
544 sctp_misc.o
545 IP_ILB_OBJS = ilb.o ilb_nat.o ilb_conn.o ilb_alg_hash.o ilb_alg_rr.o
546 IP_DCCP_OBJS = dccp.o dccp_bind.o dccp_input.o dccp_opt_data.o dccp_output.o \
547 dccp_stats.o dccp_socket.o dccp_tpi.o dccp_tunables.o
548 #endif /* ! codereview */

550 IP_OBJS +=
551 igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
552 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
553 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
554 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
555 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_queue.o \
556 squeue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
557 ip_helper_stream.o ip_tunables.o \
558 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
559 conn_opt.o ip_attr.o ip_dce.o \
560 $(IP_ICMP_OBJS) \
561 $(IP_RTS_OBJS) \
562 $(IP_TCP_OBJS) \
563 $(IP_UDP_OBJS) \
564 $(IP_SCTP_OBJS) \
565 $(IP_ILB_OBJS) \
566 $(IP_DCCP_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 sockscctp.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 += genty.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_srqp.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_srqp.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_srqp.o hermon_stats.o hermon_umap.o hermon_wr.o \
714                      hermon_fcobib.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 L1OBSERV_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_v412.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 +=      hcidi.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_uugen.o whcdi.o wa.o
836 USBA_WITHOUT_WUSB_OBJS +=      hcidi.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_uugen.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_async.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 +=      hcil1394.o hcil1394_async.o hcil1394_attach.o hcil1394_buf.o \
852                  hcil1394_csr.o hcil1394_detach.o hcil1394_extern.o \

```

```

853                  hcil1394_ioctl.o hcil1394_isoch.o hcil1394_isr.o \
854                  hcil1394_ixl_comp.o hcil1394_ixl_isr.o hcil1394_ixl_misc.o \
855                  hcil1394_ixl_update.o hcil1394_misc.o hcil1394_ohci.o \
856                  hcil1394_q.o hcil1394_s1394if.o hcil1394_tlabel.o \
857                  hcil1394_tlist.o hcil1394_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 \

```



```

919         nsc_gen.o \
920         nsc_mem.o \
921         nsc_ncallio.o \
922         nsc_power.o \
923         nsc_resv.o \
924         nsc_rmspin.o \
925         nsc_solaris.o \
926         nsc_trap.o \
927         nsc_list.o
928 UNISTAT_OBJS += spuni.o \
929         spcs_s_k.o

931 NSKERN_OBJS += nsc_ddi.o \
932         nsc_proc.o \
933         nsc_raw.o \
934         nsc_thread.o \
935         nskernd.o

937 SV_OBJS += sv.o

939 PMCS_OBJS += pmcs_attach.o pmcs_ds.o pmcs_intr.o pmcs_nvram.o pmcs_sata.o \
940         pmcs_scsa.o pmcs_smhba.o pmcs_subr.o pmcs_fwlog.o

942 PMCS8001FW_C_OBJS += pmcs_fw_hdr.o
943 PMCS8001FW_OBJS += $(PMCS8001FW_C_OBJS) SPCBoot.o ila.o firmware.o

945 #
946 #       Build up defines and paths.

948 ST_OBJS += st.o st_conf.o

950 EMLXS_OBJS += emlxs_clock.o emlxs_dfc.o emlxs_dhchap.o emlxs_diag.o \
951         emlxs_download.o emlxs_dump.o emlxs_els.o emlxs_event.o \
952         emlxs_fcf.o emlxs_fcp.o emlxs_fct.o emlxs_hba.o emlxs_ip.o \
953         emlxs_mbox.o emlxs_mem.o emlxs_msg.o emlxs_node.o \
954         emlxs_pkt.o emlxs_sli3.o emlxs_sli4.o emlxs_solaris.o \
955         emlxs_thread.o

957 EMLXS_FW_OBJS += emlxs_fw.o

959 OCE_OBJS += oce_buf.o oce_fm.o oce_gld.o oce_hw.o oce_intr.o oce_main.o \
960         oce_mbx.o oce_mq.o oce_queue.o oce_rx.o oce_stat.o oce_tx.o \
961         oce_utils.o

963 FCT_OBJS += discovery.o fct.o

965 QLT_OBJS += 2400.o 2500.o 8100.o qlt.o qlt_dma.o

967 SRPT_OBJS += srpt_mod.o srpt_ch.o srpt_cm.o srpt_ioc.o srpt_stp.o

969 FCOE_OBJS += fcoe.o fcoe_eth.o fcoe_fc.o

971 FCOET_OBJS += fcoet.o fcoet_eth.o fcoet_fc.o

973 FCOEI_OBJS += fcoei.o fcoei_eth.o fcoei_lv.o

975 ISCSIT_SHARED_OBJS += \
976         iscsit_common.o

978 ISCSIT_OBJS += $(ISCSIT_SHARED_OBJS) \
979         iscsit.o iscsit_tgt.o iscsit_sess.o iscsit_login.o \
980         iscsit_text.o iscsit_isns.o iscsit_radiusauth.o \
981         iscsit_radiuspacket.o iscsit_auth.o iscsit_authclient.o

983 PPPT_OBJS += alua_ic_if.o pppt.o pppt_msg.o pppt_tgt.o

```

```

985 STMF_OBJS += lun_map.o stmf.o

987 STMF_SBD_OBJS += sbd.o sbd_scsi.o sbd_pgr.o sbd_zvol.o

989 SYMSMSG_OBJS += sysmsg.o

991 SES_OBJS += ses.o ses_sen.o ses_safte.o ses_ses.o

993 TNF_OBJS += tnf_buf.o tnf_trace.o tnf_writer.o trace_init.o \
994         trace_funcs.o tnf_probe.o tnf.o

996 LOGINDMUX_OBJS += logindmux.o

998 DEVINFO_OBJS += devinfo.o

1000 DEVPOLL_OBJS += devpoll.o

1002 DEVPOOL_OBJS += devpool.o

1004 I8042_OBJS += i8042.o

1006 KB8042_OBJS += \
1007         at_keyprocess.o \
1008         kb8042.o \
1009         kb8042_keytables.o

1011 MOUSE8042_OBJS += mouse8042.o

1013 FDC_OBJS += fdc.o

1015 ASY_OBJS += asy.o

1017 ECPP_OBJS += ecpp.o

1019 VUIDM3P_OBJS += vuidmice.o vuidm3p.o

1021 VUIDM4P_OBJS += vuidmice.o vuidm4p.o

1023 VUIDM5P_OBJS += vuidmice.o vuidm5p.o

1025 VUIDPS2_OBJS += vuidmice.o vuidps2.o

1027 HPCSV_C_OBJS += hpcsvc.o

1029 PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehpc.o pcishpc.o pcie_pwr.o p

1031 PCIHPNEXUS_OBJS += pcihp.o

1033 OPENEEP_OBJS += openprom.o

1035 RANDOM_OBJS += random.o

1037 PSHOT_OBJS += pshot.o

1039 GEN_DRV_OBJS += gen_drv.o

1041 TCLIENT_OBJS += tclient.o

1043 TPHCI_OBJS += tphci.o

1045 TVHCI_OBJS += tvhci.o

1047 EMUL64_OBJS += emul64.o emul64_bsd.o

1049 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_pciart.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_marshallng.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 priocntl.o prsubr.o prusr.o \
1269         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 socktapi.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_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     dbuf.o \
1343     ddt.o \
1344     ddt_zap.o \
1345     dmuf.o \
1346     dmuf_diff.o \
1347     dmuf_send.o \
1348     dmuf_object.o \
1349     dmuf_objset.o \
1350     dmuf_traverse.o \
1351     dmuf_tx.o \
1352     dnode.o \
1353     dnode_sync.o \
1354     dsl_dir.o \
1355     dsl_dataset.o \
1356     dsl_deadlist.o \
1357     dsl_pool.o \
1358     dsl_synctask.o \
1359     dmuf_zfetch.o \
1360     dsl_deleg.o \
1361     dsl_prop.o \
1362     dsl_scan.o \
1363     gzip.o \
1364     lzjb.o \
1365     metaslab.o \
1366     refcount.o \
1367     sa.o \
1368     sha256.o \
1369     spa.o \
1370     spa_config.o \
1371     spa_errlog.o \
1372     spa_history.o \
1373     spa_misc.o \
1374     space_map.o \
1375     txg.o \
1376     uberblock.o \
1377     unique.o \
1378     vdev.o \
1379     vdev_cache.o \
1380     vdev_file.o \

```

```

1381     vdev_label.o \
1382     vdev_mirror.o \
1383     vdev_missing.o \
1384     vdev_queue.o \
1385     vdev_raidz.o \
1386     vdev_root.o \
1387     zap.o \
1388     zap_leaf.o \
1389     zap_micro.o \
1390     zfs_byteswap.o \
1391     zfs_debug.o \
1392     zfs_fm.o \
1393     zfs_fuid.o \
1394     zfs_sa.o \
1395     zfs_znode.o \
1396     zil.o \
1397     zio.o \
1398     zio_checksum.o \
1399     zio_compress.o \
1400     zio_inject.o \
1401     zle.o \
1402     zrlock.o

1404 ZFS_SHARED_OBJS += \
1405     zfs_namecheck.o \
1406     zfs_deleg.o \
1407     zfs_prop.o \
1408     zfs_comutil.o \
1409     zfs_fletcher.o \
1410     zpool_prop.o \
1411     zprop_common.o

1413 ZFS_OBJS += \
1414     $(ZFS_COMMON_OBJS) \
1415     $(ZFS_SHARED_OBJS) \
1416     vdev_disk.o \
1417     zfs_acl.o \
1418     zfs_ctldir.o \
1419     zfs_dir.o \
1420     zfs_ioctl.o \
1421     zfs_log.o \
1422     zfs_onexit.o \
1423     zfs_replay.o \
1424     zfs_rlock.o \
1425     rrwlock.o \
1426     zfs_vfsops.o \
1427     zfs_vnops.o \
1428     zvol.o

1430 ZUT_OBJS += \
1431     zut.o

1433 #
1434 #                streams modules
1435 #
1436 BUFMOD_OBJS += bufmod.o

1438 CONNLD_OBJS += connld.o

1440 DEDUMP_OBJS += dedump.o

1442 DRCOMPAT_OBJS += drcompat.o

1444 LDLINUX_OBJS += ldlinux.o

1446 LDTERM_OBJS += ldterm.o uwidth.o

```

```

1448 PKCT_OBJS +=      pckt.o
1450 PFMOD_OBJS +=      pfmod.o
1452 PTEM_OBJS +=      ptem.o
1454 REDIRMOD_OBJS +=  strredirm.o
1456 TIMOD_OBJS +=      timod.o
1458 TIRDWR_OBJS +=     tirdwr.o
1460 TTCOMPAT_OBJS +=  ttcompat.o
1462 LOG_OBJS +=        log.o
1464 PIPEMOD_OBJS +=    pipemod.o

1466 RPCMOD_OBJS +=      rpcmod.o      clnt_cots.o      clnt_clts.o \
1467                       clnt_gen.o      clnt_perr.o      mt_rpcinit.o      rpc_calmsg.o \
1468                       rpc_prot.o      rpc_sztypes.o   rpc_subr.o        rpcb_prot.o \
1469                       svc.o           svc_clts.o      svc_gen.o         svc_cots.o \
1470                       rpcsys.o        xdr_sizeof.o   clnt_rdma.o      svc_rdma.o \
1471                       xdr_rdma.o      rdma_subr.o    xdrdma_sizeof.o

1473 TLIMOD_OBJS +=      tlimod.o      t_kalloc.o      t_kbind.o      t_kclose.o \
1474                       t_kconnect.o    t_kfree.o       t_kgtstate.o   t_kopen.o \
1475                       t_krcvudat.o    t_ksndudat.o   t_kspoll.o     t_kunbind.o \
1476                       t_kutil.o

1478 RLMOD_OBJS +=      rlmmod.o
1480 TELMOD_OBJS +=     telmod.o
1482 CRYPTMOD_OBJS +=   cryptmod.o
1484 KB_OBJS +=         kbd.o           keytables.o

1486 #
1487 #             ID mapping module
1488 #
1489 IDMAP_OBJS +=      idmap_mod.o      idmap_kapi.o    idmap_xdr.o     idmap_cache.o

1491 #
1492 #             scheduling class modules
1493 #
1494 SDC_OBJS +=        sysdc.o

1496 RT_OBJS +=         rt.o
1497 RT_DPTBL_OBJS +=   rt_dptbl.o

1499 TS_OBJS +=         ts.o
1500 TS_DPTBL_OBJS +=   ts_dptbl.o

1502 IA_OBJS +=        ia.o
1504 FSS_OBJS +=       fss.o

1506 FX_OBJS +=         fx.o
1507 FX_DPTBL_OBJS +=   fx_dptbl.o

1509 #
1510 #             Inter-Process Communication (IPC) modules
1511 #
1512 IPC_OBJS +=         ipc.o

```

```

1514 IPCMSG_OBJS +=     msg.o
1516 IPCSEM_OBJS +=     sem.o
1518 IPCSHM_OBJS +=     shm.o

1520 #
1521 #             bignum module
1522 #
1523 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o

1525 BIGNUM_OBJS +=     $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)

1527 #
1528 #             kernel cryptographic framework
1529 #
1530 KCF_OBJS +=         kcf.o kcf_callprov.o kcf_cbufcall.o kcf_cipher.o kcf_crypto.o \
1531                       kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1532                       kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1533                       kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1534                       kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1535                       kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1536                       fips_random.o

1538 CRYPTOADM_OBJS +=   cryptoadm.o
1540 CRYPTO_OBJS +=      crypto.o
1542 DPROV_OBJS +=       dprov.o

1544 DCA_OBJS +=          dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1545                       dca_rsa.o

1547 AESPROV_OBJS +=     aes.o aes_impl.o aes_modes.o
1549 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o

1551 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o

1553 ECCPROV_OBJS +=     ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1554                       ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1555                       ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1556                       ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1557                       mpi.o mplogic.o mpmontg.o mpprime.o oid.o \
1558                       secitem.o ec2_test.o ecp_test.o

1560 RSAPROV_OBJS +=     rsa.o rsa_impl.o pkcs1.o

1562 SWRANDPROV_OBJS += swrand.o

1564 #
1565 #             kernel SSL
1566 #
1567 KSSL_OBJS +=         kssl.o ksslioct1.o

1569 KSSL_SOCKETFIL_MOD_OBJS += ksslfilter.o ksslapi.o ksslrec.o

1571 #
1572 #             misc. modules
1573 #

1575 C2AUDIT_OBJS +=     adr.o audit.o audit_event.o audit_io.o \
1576                       audit_path.o audit_start.o audit_syscalls.o audit_token.o \
1577                       audit_mem.o

```

```

1579 PCIC_OBJS += pcic.o

1581 RPCSEC_OBJS += secmod.o sec_clnt.o sec_svc.o sec_gen.o \
1582 auth_des.o auth_kern.o auth_none.o auth_loopb.o \
1583 authdesprt.o authdesubr.o authu_prot.o \
1584 key_call.o key_prot.o svc_authu.o svcauthdes.o

1586 RPCSEC_GSS_OBJS += rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
1587 rpcsec_gss_utils.o svc_rpcsec_gss.o

1589 CONSCONFIG_OBJS += consconfig.o

1591 CONSCONFIG_DACF_OBJS += consconfig_dacf.o consplat.o

1593 TEM_OBJS += tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1595 KBTRANS_OBJS += \
1596 kbtrans.o \
1597 kbtrans_keytables.o \
1598 kbtrans_polled.o \
1599 kbtrans_streams.o \
1600 usb_keytables.o

1602 KGSSD_OBJS += gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1603 gss_display_name.o gss_release_name.o gss_import_name.o \
1604 gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1606 KGSSD_DERIVED_OBJS = gssd_xdr.o

1608 KGSS_DUMMY_OBJS += dmech.o

1610 KSOCKET_OBJS += ksocket.o ksocket_mod.o

1612 CRYPTO= cksumentypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1613 nfold.o verify_checksum.o prng.o block_size.o make_checksum.o \
1614 checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1616 # crypto/des
1617 CRYPTO_DES= f_cbc.o f_cksum.o f_parity.o weak_key.o d3_cbc.o ef_crypto.o

1619 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1621 CRYPTO_ARCFOUR= k5_arcfour.o

1623 # crypto/enc_provider
1624 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o

1626 # crypto/hash_provider
1627 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_ksha1.o

1629 # crypto/keyhash_provider
1630 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1632 # crypto/crc32
1633 CRYPTO_CRC32= crc32.o

1635 # crypto/old
1636 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1638 # crypto/raw
1639 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1641 K5_KRB= kfree.o copy_key.o \
1642 parse.o init_ctx.o \
1643 ser_adata.o ser_addr.o \
1644 ser_auth.o ser_cksum.o \

```

```

1645 ser_key.o ser_princ.o \
1646 serialize.o unparse.o \
1647 ser_actx.o

1649 K5_OS= timeofday.o toffset.o \
1650 init_os_ctx.o c_ustime.o

1652 SEAL=
1653 # EXPORT DELETE START
1654 SEAL= seal.o unseal.o
1655 # EXPORT DELETE END

1657 MECH= delete_sec_context.o \
1658 import_sec_context.o \
1659 gssapi_krb5.o \
1660 k5seal.o k5unseal.o k5sealv3.o \
1661 ser_sctx.o \
1662 sign.o \
1663 util_crypt.o \
1664 util_validate.o util_ordering.o \
1665 util_seqnum.o util_set.o util_seed.o \
1666 wrap_size_limit.o verify.o

1670 MECH_GEN= util_token.o

1673 KGSS_KRB5_OBJS += krb5mech.o \
1674 $(MECH) $(SEAL) $(MECH_GEN) \
1675 $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1676 $(CRYPTO_ENC) $(CRYPTO_HASH) \
1677 $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1678 $(CRYPTO_OLD) \
1679 $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1681 DES_OBJS += des_crypt.o des_impl.o des_ks.o des_soft.o

1683 DLBOOT_OBJS += bootparam_xdr.o nfs_dlinet.o scan.o

1685 KRTLD_OBJS += kobj_bootflags.o getoptstr.o \
1686 kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1688 MOD_OBJS += modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1690 STRPLUMB_OBJS += strplumb.o

1692 CPR_OBJS += cpr_driver.o cpr_dump.o \
1693 cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1694 cpr_uthread.o

1696 PROF_OBJS += prf.o

1698 SE_OBJS += se_driver.o

1700 SYSACCT_OBJS += acct.o

1702 ACCTCTL_OBJS += acctctl.o

1704 EXACCTSYS_OBJS += exacctsys.o

1706 KAIIO_OBJS += aio.o

1708 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o

1710 BUSRA_OBJS += busra.o

```

```

1712 PCS_OBJS += pcs.o
1714 PCAN_OBJS += pcan.o
1716 PCATA_OBJS += pcide.o pcdisk.o pclabel.o pcata.o
1718 PCSER_OBJS += pcser.o pcser_cis.o
1720 PCWL_OBJS += pcwl.o
1722 PSET_OBJS += pset.o
1724 OHCI_OBJS += ohci.o ohci_hub.o ohci_polled.o
1726 UHCI_OBJS += uhci.o uhciutil.o uhcitgt.o uhcihub.o uhcipolled.o
1728 EHCI_OBJS += ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o
1730 HUBD_OBJS += hubd.o
1732 USB_MID_OBJS += usb_mid.o
1734 USB_IA_OBJS += usb_ia.o
1736 UWBA_OBJS += uwba.o uwbai.o
1738 SCSA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o
1740 HWAHC_OBJS += hwahc.o hwahc_util.o
1742 WUSB_DF_OBJS += wusb_df.o
1743 WUSB_FWMOD_OBJS += wusb_fwmod.o
1745 IPF_OBJS += ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1746 ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1747 ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o
1749 IBD_OBJS += ibd.o ibd_cm.o
1751 EIBNX_OBJS += enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1752 enx_misc.o enx_q.o enx_ctl.o
1754 EOIB_OBJS += eib_adm.o eib_chan.o eib_cmn.o eib_ctl.o eib_data.o \
1755 eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1756 eib_rsrc.o eib_svc.o eib_vnic.o
1758 DLPSTUB_OBJS += dlpistub.o
1760 SDP_OBJS += sdpddi.o
1762 TRILL_OBJS += trill.o
1764 CTF_OBJS += ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1765 ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o
1767 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o
1769 RPCIB_OBJS += rpcib.o
1771 KMDB_OBJS += kdrv.o
1773 AFE_OBJS += afe.o
1775 BGE_OBJS += bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1776 bge_atomic.o bge_mii.o bge_send.o bge_recv2.o bge_mii_5906.o

```

```

1778 DMFE_OBJS += dmfe_log.o dmfe_main.o dmfe_mii.o
1780 EFE_OBJS += efe.o
1782 ELXL_OBJS += elxl.o
1784 HME_OBJS += hme.o
1786 IXGB_OBJS += ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1787 ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o
1789 NGE_OBJS += nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1790 nge_log.o nge_rx.o nge_tx.o nge_xmii.o
1792 PCN_OBJS += pcn.o
1794 RGE_OBJS += rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rtx.o
1796 URTW_OBJS += urtw.o
1798 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_ani.o arn_phy.o arn_
1799 arn_main.o arn_recv.o arn_xmit.o arn_rc.o
1801 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o
1803 ATU_OBJS += atu.o
1805 IPW_OBJS += ipw2100_hw.o ipw2100.o
1807 IWI_OBJS += ipw2200_hw.o ipw2200.o
1809 IWH_OBJS += iwh.o
1811 IWK_OBJS += iwk2.o
1813 IWP_OBJS += iwp.o
1815 MWL_OBJS += mwl.o
1817 MWLFW_OBJS += mwlfw_mode.o
1819 WPI_OBJS += wpi.o
1821 RAL_OBJS += rt2560.o ral_rate.o
1823 RUM_OBJS += rum.o
1825 RWD_OBJS += rt2661.o
1827 RWN_OBJS += rt2860.o
1829 UATH_OBJS += uath.o
1831 UATHFW_OBJS += uathfw_mod.o
1833 URAL_OBJS += ural.o
1835 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o
1837 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o
1839 MXFE_OBJS += mxfe.o
1841 MPTSAS_OBJS += mptsas.o mptsas_impl.o mptsas_init.o mptsas_raid.o mptsas_smhba.o

```

```

1843 SFE_OBJS += sfe.o sfe_util.o
1845 BFE_OBJS += bfe.o
1847 BRIDGE_OBJS += bridge.o
1849 IDM_SHARED_OBJS += base64.o
1851 IDM_OBJS += $(IDM_SHARED_OBJS) \
1852         idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o
1854 VR_OBJS += vr.o
1856 ATGE_OBJS += atge_main.o atge_lle.o atge_mii.o atge_ll.o
1858 YGE_OBJS = yge.o
1860 #
1861 #     Build up defines and paths.
1862 #
1863 LINT_DEFS     += -Dunix
1865 #
1866 #     This duality can be removed when the native and target compilers
1867 #     are the same (or at least recognize the same command line syntax!)
1868 #     It is a bug in the current compilation system that the assembler
1869 #     can't process the -Y I, flag.
1870 #
1871 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1872 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1873 INCLUDE_PATH   += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1875 PCIEB_OBJS += pcieb.o
1877 #     Chelsio N110 10G NIC driver module
1878 #
1879 CH_OBJS = ch.o glue.o pe.o sge.o
1881 CH_COM_OBJS = ch_mac.o ch_subr.o cspi.o espi.o ixfl1010.o mc3.o mc4.o mc5.o \
1882         mv88elxxx.o mv88x20lx.o my3126.o pm3393.o tp.o ulp.o \
1883         vsc7321.o vsc7326.o xpak.o
1885 #
1886 #     PCI strings file
1887 #
1888 PCI_STRING_OBJS = pci_strings.o
1890 NET_DACF_OBJS += net_dacf.o
1892 #
1893 #     Xframe 10G NIC driver module
1894 #
1895 XGE_OBJS = xge.o xgell.o
1897 XGE_HAL_OBJS = xgehal-channel.o xgehal-fifo.o xgehal-ring.o xgehal-config.o \
1898         xgehal-driver.o xgehal-mm.o xgehal-stats.o xgehal-device.o \
1899         xge-queue.o xgehal-mgmt.o xgehal-mgmtaux.o
1901 #
1902 #     e1000g module
1903 #
1904 E1000G_OBJS += e1000_80003es2lan.o e1000_82540.o e1000_82541.o e1000_82542.o \
1905         e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
1906         e1000_mac.o e1000_manage.o e1000_nvmm.o e1000_osdep.o \
1907         e1000_phy.o e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1908         e1000g_tx.o e1000g_rx.o e1000g_stat.o

```

```

1910 #
1911 #     Intel 82575 1G NIC driver module
1912 #
1913 IGB_OBJS = igb_82575.o igb_api.o igb_mac.o igb_manage.o \
1914         igb_nvmm.o igb_osdep.o igb_phy.o igb_buf.o \
1915         igb_debug.o igb_gld.o igb_log.o igb_main.o \
1916         igb_rx.o igb_stat.o igb_tx.o
1918 #
1919 #     Intel Pro/100 NIC driver module
1920 #
1921 IPRB_OBJS = iprb.o
1923 #
1924 #     Intel 10GbE PCIE NIC driver module
1925 #
1926 IXGBE_OBJS = ixgbe_82598.o ixgbe_82599.o ixgbe_api.o \
1927         ixgbe_common.o ixgbe_phy.o \
1928         ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o \
1929         ixgbe_log.o ixgbe_main.o \
1930         ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o \
1931         ixgbe_tx.o
1933 #
1934 #     NIU 10G/1G driver module
1935 #
1936 NXGE_OBJS = nxge_mac.o nxge_ipp.o nxge_rxdma.o \
1937         nxge_txdma.o nxge_txc.o nxge_main.o \
1938         nxge_hw.o nxge_fzc.o nxge_virtual.o \
1939         nxge_send.o nxge_classify.o nxge_fflp.o \
1940         nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o \
1941         nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o \
1942         nxge_hio.o nxge_hio_guest.o nxge_intr.o
1944 NXGE_NPI_OBJS = \
1945         np_i.o np_i_mac.o np_i_ipp.o \
1946         np_i_txdma.o np_i_rxdma.o np_i_txc.o \
1947         np_i_zcp.o np_i_espc.o np_i_fflp.o \
1948         np_i_vir.o
1950 NXGE_HCALL_OBJS = \
1951         nxge_hcall.o
1953 #
1954 #     kiconv modules
1955 #
1956 KICONV_EMEA_OBJS += kiconv_emea.o
1958 KICONV_JA_OBJS += kiconv_ja.o
1960 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o
1962 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o
1964 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o
1966 #
1967 #     AAC module
1968 #
1969 AAC_OBJS = aac.o aac_ioctl.o
1971 #
1972 #     sdcard modules
1973 #
1974 SDA_OBJS = sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o

```



```
1975 SDHOST_OBJS = sdhost.o

1977 #
1978 #     hxge 10G driver module
1979 #
1980 HXGE_OBJS =     hxge_main.o hxge_vmac.o hxge_send.o     \
1981             hxge_txdma.o hxge_rxdma.o hxge_virtual.o   \
1982             hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o \
1983             hxge_ndd.o hxge_pfc.o                     \
1984             hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o    \
1985             hpi_vir.o hpi_pfc.o

1987 #
1988 #     MEGARAID_SAS module
1989 #
1990 MEGA_SAS_OBJS = megaraid_sas.o

1992 #
1993 #     MR_SAS module
1994 #
1995 MR_SAS_OBJS = mr_sas.o

1997 #
1998 #     ISCSI_INITIATOR module
1999 #
2000 ISCSI_INITIATOR_OBJS = chap.o iscsi_io.o iscsi_thread.o \
2001                       iscsi_ioctl.o iscsid.o iscsi.o   \
2002                       iscsi_login.o isns_client.o iscsiAuthClient.o \
2003                       iscsi_lun.o iscsiAuthClientGlue.o \
2004                       iscsi_net.o nvfile.o iscsi_cmd.o  \
2005                       iscsi_queue.o persistent.o iscsi_conn.o \
2006                       iscsi_sess.o radius_auth.o iscsi_crc.o \
2007                       iscsi_stats.o radius_packet.o iscsi_doorclt.o \
2008                       iscsi_targetparam.o utils.o kifconf.o

2010 #
2011 #     ntxn 10Gb/1Gb NIC driver module
2012 #
2013 NTXN_OBJS =     unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
2014               unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

2016 #
2017 #     Myricom 10Gb NIC driver module
2018 #
2019 MYRI10GE_OBJS = myri10ge.o myri10ge_lro.o

2021 #
2022 #     nulldriver module
2023 NULLDRIVER_OBJS =     nulldriver.o

2025 TPM_OBJS =     tpm.o tpm_hcall.o
```

```

*****
72399 Wed Jun 13 12:04:24 2012
new/usr/src/uts/common/Makefile.rules
%B
*****
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 #
21 #
22 #
23 # Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
24 #
25 #
26 #
27 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
28 #
29 #
30 #
31 # uts/common/Makefile.rules
32 #
33 # This Makefile defines all the file build rules for the directory
34 # uts/common and its children. These are the source files which may
35 # be considered common to all SunOS systems.
36 #
37 # The following two-level ordering must be maintained in this file.
38 # Lines are sorted first in order of decreasing specificity based on
39 # the first directory component. That is, sun4u rules come before
40 # sparc rules come before common rules.
41 #
42 # Lines whose initial directory components are equal are sorted
43 # alphabetically by the remaining components.
44 #
45 #
46 # Section 1a: C objects build rules
47 #
48 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/aes/%.c
49 $(COMPILE.c) -o $@ $<
50 $(CTFCONVERT_O)
51 #
52 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/arcfour/%.c
53 $(COMPILE.c) -o $@ $<
54 $(CTFCONVERT_O)
55 #
56 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/blowfish/%.c
57 $(COMPILE.c) -o $@ $<
58 $(CTFCONVERT_O)
59 #
60 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/ecc/%.c
61 $(COMPILE.c) -o $@ $<

```

```

62 $(CTFCONVERT_O)
63 #
64 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/modes/%.c
65 $(COMPILE.c) -o $@ $<
66 $(CTFCONVERT_O)
67 #
68 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/padding/%.c
69 $(COMPILE.c) -o $@ $<
70 $(CTFCONVERT_O)
71 #
72 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/rng/%.c
73 $(COMPILE.c) -o $@ $<
74 $(CTFCONVERT_O)
75 #
76 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/rsa/%.c
77 $(COMPILE.c) -o $@ $<
78 $(CTFCONVERT_O)
79 #
80 $(OBJSDIR)/%.o: $(COMMONBASE)/bignum/%.c
81 $(COMPILE.c) -o $@ $<
82 $(CTFCONVERT_O)
83 #
84 $(OBJSDIR)/%.o: $(UTSBASE)/common/bignum/%.c
85 $(COMPILE.c) -o $@ $<
86 $(CTFCONVERT_O)
87 #
88 $(OBJSDIR)/%.o: $(COMMONBASE)/mpi/%.c
89 $(COMPILE.c) -o $@ $<
90 $(CTFCONVERT_O)
91 #
92 $(OBJSDIR)/%.o: $(COMMONBASE)/acl/%.c
93 $(COMPILE.c) -o $@ $<
94 $(CTFCONVERT_O)
95 #
96 $(OBJSDIR)/%.o: $(COMMONBASE)/avl/%.c
97 $(COMPILE.c) -o $@ $<
98 $(CTFCONVERT_O)
99 #
100 $(OBJSDIR)/%.o: $(COMMONBASE)/ucode/%.c
101 $(COMPILE.c) -o $@ $<
102 $(CTFCONVERT_O)
103 #
104 $(OBJSDIR)/%.o: $(UTSBASE)/common/brand/sn1/%.c
105 $(COMPILE.c) -o $@ $<
106 $(CTFCONVERT_O)
107 #
108 $(OBJSDIR)/%.o: $(UTSBASE)/common/brand/solaris10/%.c
109 $(COMPILE.c) -o $@ $<
110 $(CTFCONVERT_O)
111 #
112 $(OBJSDIR)/%.o: $(UTSBASE)/common/c2/%.c
113 $(COMPILE.c) -o $@ $<
114 $(CTFCONVERT_O)
115 #
116 $(OBJSDIR)/%.o: $(UTSBASE)/common/conf/%.c
117 $(COMPILE.c) -o $@ $<
118 $(CTFCONVERT_O)
119 #
120 $(OBJSDIR)/%.o: $(UTSBASE)/common/contract/%.c
121 $(COMPILE.c) -o $@ $<
122 $(CTFCONVERT_O)
123 #
124 $(OBJSDIR)/%.o: $(UTSBASE)/common/cpr/%.c
125 $(COMPILE.c) -o $@ $<
126 $(CTFCONVERT_O)

```

new/usr/src/uts/common/Makefile.rules

```

128 $(OBJSDIR)/%.o: $(UTSBASE)/common/ctf/%.c
129 $(COMPILE.c) -o $@ $<
130 $(CTFCONVERT_O)

132 $(OBJSDIR)/%.o: $(COMMONBASE)/ctf/%.c
133 $(COMPILE.c) -o $@ $<
134 $(CTFCONVERT_O)

136 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/des/%.c
137 $(COMPILE.c) -o $@ $<
138 $(CTFCONVERT_O)

140 $(OBJSDIR)/%.o: $(COMMONBASE)/smbios/%.c
141 $(COMPILE.c) -o $@ $<
142 $(CTFCONVERT_O)

144 $(OBJSDIR)/%.o: $(UTSBASE)/common/des/%.c
145 $(COMPILE.c) -o $@ $<
146 $(CTFCONVERT_O)

148 $(OBJSDIR)/%.o: $(UTSBASE)/common/crypto/api/%.c
149 $(COMPILE.c) -o $@ $<
150 $(CTFCONVERT_O)

152 $(OBJSDIR)/%.o: $(UTSBASE)/common/crypto/core/%.c
153 $(COMPILE.c) -o $@ $<
154 $(CTFCONVERT_O)

156 $(OBJSDIR)/%.o: $(UTSBASE)/common/crypto/io/%.c
157 $(COMPILE.c) -o $@ $<
158 $(CTFCONVERT_O)

160 $(OBJSDIR)/%.o: $(UTSBASE)/common/crypto/spi/%.c
161 $(COMPILE.c) -o $@ $<
162 $(CTFCONVERT_O)

164 $(OBJSDIR)/%.o: $(COMMONBASE)/pci/%.c
165 $(COMPILE.c) -o $@ $<
166 $(CTFCONVERT_O)

168 $(OBJSDIR)/%.o: $(COMMONBASE)/devid/%.c
169 $(COMPILE.c) -o $@ $<
170 $(CTFCONVERT_O)

172 $(OBJSDIR)/%.o: $(UTSBASE)/common/disp/%.c
173 $(COMPILE.c) -o $@ $<
174 $(CTFCONVERT_O)

176 $(OBJSDIR)/%.o: $(UTSBASE)/common/dtrace/%.c
177 $(COMPILE.c) -o $@ $<
178 $(CTFCONVERT_O)

180 $(OBJSDIR)/%.o: $(COMMONBASE)/execct/%.c
181 $(COMPILE.c) -o $@ $<
182 $(CTFCONVERT_O)

184 $(OBJSDIR)/%.o: $(UTSBASE)/common/exec/aout/%.c
185 $(COMPILE.c) -o $@ $<
186 $(CTFCONVERT_O)

188 $(OBJSDIR)/%.o: $(UTSBASE)/common/exec/elf/%.c
189 $(COMPILE.c) -o $@ $<
190 $(CTFCONVERT_O)

192 $(OBJSDIR)/%.o: $(UTSBASE)/common/exec/intp/%.c
193 $(COMPILE.c) -o $@ $<

```

3

new/usr/src/uts/common/Makefile.rules

```

194 $(CTFCONVERT_O)

196 $(OBJSDIR)/%.o: $(UTSBASE)/common/exec/shbin/%.c
197 $(COMPILE.c) -o $@ $<
198 $(CTFCONVERT_O)

200 $(OBJSDIR)/%.o: $(UTSBASE)/common/exec/java/%.c
201 $(COMPILE.c) -o $@ $<
202 $(CTFCONVERT_O)

204 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/%.c
205 $(COMPILE.c) -o $@ $<
206 $(CTFCONVERT_O)

208 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/autofs/%.c
209 $(COMPILE.c) -o $@ $<
210 $(CTFCONVERT_O)

212 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/cacheofs/%.c
213 $(COMPILE.c) -o $@ $<
214 $(CTFCONVERT_O)

216 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/dcfis/%.c
217 $(COMPILE.c) -o $@ $<
218 $(CTFCONVERT_O)

220 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/devfs/%.c
221 $(COMPILE.c) -o $@ $<
222 $(CTFCONVERT_O)

224 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/ctfs/%.c
225 $(COMPILE.c) -o $@ $<
226 $(CTFCONVERT_O)

228 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/doorfs/%.c
229 $(COMPILE.c) -o $@ $<
230 $(CTFCONVERT_O)

232 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/dev/%.c
233 $(COMPILE.c) -o $@ $<
234 $(CTFCONVERT_O)

236 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/fd/%.c
237 $(COMPILE.c) -o $@ $<
238 $(CTFCONVERT_O)

240 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/fifoofs/%.c
241 $(COMPILE.c) -o $@ $<
242 $(CTFCONVERT_O)

244 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/hsfs/%.c
245 $(COMPILE.c) -o $@ $<
246 $(CTFCONVERT_O)

248 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/lofs/%.c
249 $(COMPILE.c) -o $@ $<
250 $(CTFCONVERT_O)

252 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/mntfs/%.c
253 $(COMPILE.c) -o $@ $<
254 $(CTFCONVERT_O)

256 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/namefs/%.c
257 $(COMPILE.c) -o $@ $<
258 $(CTFCONVERT_O)

```

4

new/usr/src/uts/common/Makefile.rules

5

```

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/pcfefs/%.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/net smb/%.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/uifs/%.c
325 $(COMPILE.c) -o $@ $<

```

new/usr/src/uts/common/Makefile.rules

6

```

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)

```

```

393 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/dk/%.c
394 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
395 $(CTFCONVERT_O)

397 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/enc_provider/%.c
398 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
399 $(CTFCONVERT_O)

401 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/hash_provider/%.c
402 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
403 $(CTFCONVERT_O)

405 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/keyhash_provider/%.c
406 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
407 $(CTFCONVERT_O)

409 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/raw/%.c
410 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
411 $(CTFCONVERT_O)

413 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/old/%.c
414 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
415 $(CTFCONVERT_O)

417 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/krb5/krb/%.c
418 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
419 $(CTFCONVERT_O)

421 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/krb5/os/%.c
422 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
423 $(CTFCONVERT_O)

425 $(OBJSDIR)/ser_sctx.o := CPPFLAGS += -DPROVIDE_KERNEL_IMPORT=1

427 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/mech/%.c
428 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
429 $(CTFCONVERT_O)

431 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/profile/%.c
432 $(COMPILE.c) $(KGSSDFLAGS) -o $$@ $$<
433 $(CTFCONVERT_O)

435 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ncall/%.c
436 $(COMPILE.c) -o $$@ $$<
437 $(CTFCONVERT_O)

439 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ns/dsw/%.c
440 $(COMPILE.c) -o $$@ $$<
441 $(CTFCONVERT_O)

443 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ns/nsctl/%.c
444 $(COMPILE.c) -o $$@ $$<
445 $(CTFCONVERT_O)

447 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ns/rdc/%.c
448 $(COMPILE.c) -o $$@ $$<
449 $(CTFCONVERT_O)

451 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ns/sdbc/%.c
452 $(COMPILE.c) -o $$@ $$<
453 $(CTFCONVERT_O)

455 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ns/solaris/%.c
456 $(COMPILE.c) -o $$@ $$<
457 $(CTFCONVERT_O)

```

```

459 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ns/sv/%.c
460 $(COMPILE.c) -o $$@ $$<
461 $(CTFCONVERT_O)

463 $(OBJSDIR)/%.o: $(UTSBASE)/common/avs/ns/unistat/%.c
464 $(COMPILE.c) -o $$@ $$<
465 $(CTFCONVERT_O)

467 $(OBJSDIR)/%.o: $(UTSBASE)/common/idmap/%.c
468 $(COMPILE.c) -o $$@ $$<
469 $(CTFCONVERT_O)

471 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/%.c
472 $(COMPILE.c) -o $$@ $$<
473 $(CTFCONVERT_O)

475 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/arp/%.c
476 $(COMPILE.c) -o $$@ $$<
477 $(CTFCONVERT_O)

479 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/dccp/%.c
480 $(COMPILE.c) -o $$@ $$<
481 $(CTFCONVERT_O)

483 #endif /* ! codereview */
484 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/ip/%.c
485 $(COMPILE.c) -o $$@ $$<
486 $(CTFCONVERT_O)

488 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/ipnet/%.c
489 $(COMPILE.c) -o $$@ $$<
490 $(CTFCONVERT_O)

492 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/iptun/%.c
493 $(COMPILE.c) -o $$@ $$<
494 $(CTFCONVERT_O)

496 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/kssl/%.c
497 $(COMPILE.c) -o $$@ $$<
498 $(CTFCONVERT_O)

500 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/sctp/%.c
501 $(COMPILE.c) -o $$@ $$<
502 $(CTFCONVERT_O)

504 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/tcp/%.c
505 $(COMPILE.c) -o $$@ $$<
506 $(CTFCONVERT_O)

508 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/ilb/%.c
509 $(COMPILE.c) -o $$@ $$<
510 $(CTFCONVERT_O)

512 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/ipf/%.c
513 $(COMPILE.c) -o $$@ $$<
514 $(CTFCONVERT_O)

516 $(OBJSDIR)/%.o: $(COMMONBASE)/net/patricia/%.c
517 $(COMPILE.c) -o $$@ $$<
518 $(CTFCONVERT_O)

520 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/udp/%.c
521 $(COMPILE.c) -o $$@ $$<
522 $(CTFCONVERT_O)

```

```

524 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/nca/%.c
525     $(COMPILE.c) -o $@ $<
526     $(CTFCONVERT_O)

528 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/sockmods/%.c
529     $(COMPILE.c) -o $@ $<
530     $(CTFCONVERT_O)

532 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/dlpistub/%.c
533     $(COMPILE.c) -o $@ $<
534     $(CTFCONVERT_O)

536 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/%.c
537     $(COMPILE.c) -o $@ $<
538     $(CTFCONVERT_O)

540 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/1394/%.c
541     $(COMPILE.c) -o $@ $<
542     $(CTFCONVERT_O)

544 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/1394/adapters/%.c
545     $(COMPILE.c) -o $@ $<
546     $(CTFCONVERT_O)

548 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/1394/targets/av1394/%.c
549     $(COMPILE.c) -o $@ $<
550     $(CTFCONVERT_O)

552 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/1394/targets/dcam1394/%.c
553     $(COMPILE.c) -o $@ $<
554     $(CTFCONVERT_O)

556 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/1394/targets/scsa1394/%.c
557     $(COMPILE.c) -o $@ $<
558     $(CTFCONVERT_O)

560 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sbp2/%.c
561     $(COMPILE.c) -o $@ $<
562     $(CTFCONVERT_O)

564 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/aac/%.c
565     $(COMPILE.c) -o $@ $<
566     $(CTFCONVERT_O)

568 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/afe/%.c
569     $(COMPILE.c) -o $@ $<
570     $(CTFCONVERT_O)

572 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/atge/%.c
573     $(COMPILE.c) -o $@ $<
574     $(CTFCONVERT_O)

576 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/arn/%.c
577     $(COMPILE.c) -o $@ $<
578     $(CTFCONVERT_O)

580 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ath/%.c
581     $(COMPILE.c) -o $@ $<
582     $(CTFCONVERT_O)

584 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/atu/%.c
585     $(COMPILE.c) -o $@ $<
586     $(CTFCONVERT_O)

588 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/impl/%.c
589     $(COMPILE.c) -o $@ $<

```

```

590     $(CTFCONVERT_O)

592 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/ac97/%.c
593     $(COMPILE.c) -o $@ $<
594     $(CTFCONVERT_O)

596 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audioens/%.c
597     $(COMPILE.c) -o $@ $<
598     $(CTFCONVERT_O)

600 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audioemul0k/%.c
601     $(COMPILE.c) -o $@ $<
602     $(CTFCONVERT_O)

604 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audio1575/%.c
605     $(COMPILE.c) -o $@ $<
606     $(CTFCONVERT_O)

608 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audio810/%.c
609     $(COMPILE.c) -o $@ $<
610     $(CTFCONVERT_O)

612 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiocmi/%.c
613     $(COMPILE.c) -o $@ $<
614     $(CTFCONVERT_O)

616 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiocmihd/%.c
617     $(COMPILE.c) -o $@ $<
618     $(CTFCONVERT_O)

620 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiohd/%.c
621     $(COMPILE.c) -o $@ $<
622     $(CTFCONVERT_O)

624 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audioixp/%.c
625     $(COMPILE.c) -o $@ $<
626     $(CTFCONVERT_O)

628 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiols/%.c
629     $(COMPILE.c) -o $@ $<
630     $(CTFCONVERT_O)

632 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiopci/%.c
633     $(COMPILE.c) -o $@ $<
634     $(CTFCONVERT_O)

636 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiopl6x/%.c
637     $(COMPILE.c) -o $@ $<
638     $(CTFCONVERT_O)

640 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiosolo/%.c
641     $(COMPILE.c) -o $@ $<
642     $(CTFCONVERT_O)

644 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiotots/%.c
645     $(COMPILE.c) -o $@ $<
646     $(CTFCONVERT_O)

648 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiovia823x/%.c
649     $(COMPILE.c) -o $@ $<
650     $(CTFCONVERT_O)

652 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/audio/drv/audiovia97/%.c
653     $(COMPILE.c) -o $@ $<
654     $(CTFCONVERT_O)

```

```

656 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/bfe/%.c
657     $(COMPILE.c) -o $@ $<
658     $(CTFCONVERT_O)

660 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/bge/%.c
661     $(COMPILE.c) -o $@ $<
662     $(CTFCONVERT_O)

664 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/blkdev/%.c
665     $(COMPILE.c) -o $@ $<
666     $(CTFCONVERT_O)

668 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/bpf/%.c
669     $(COMPILE.c) -o $@ $<
670     $(CTFCONVERT_O)

672 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/cardbus/%.c
673     $(COMPILE.c) -o $@ $<
674     $(CTFCONVERT_O)

676 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/stmf/%.c
677     $(COMPILE.c) -o $@ $<
678     $(CTFCONVERT_O)

680 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/port/fct/%.c
681     $(COMPILE.c) -o $@ $<
682     $(CTFCONVERT_O)

684 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/port/qlt/%.c
685     $(COMPILE.c) -o $@ $<
686     $(CTFCONVERT_O)

688 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/port/srpt/%.c
689     $(COMPILE.c) -o $@ $<
690     $(CTFCONVERT_O)

692 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/port/fcoet/%.c
693     $(COMPILE.c) -o $@ $<
694     $(CTFCONVERT_O)

696 $(OBJSDIR)/%.o: $(COMMONBASE)/iscsit/%.c
697     $(COMPILE.c) -o $@ $<
698     $(CTFCONVERT_O)

700 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/port/iscsit/%.c
701     $(COMPILE.c) -o $@ $<
702     $(CTFCONVERT_O)

704 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/port/pppt/%.c
705     $(COMPILE.c) -o $@ $<
706     $(CTFCONVERT_O)

708 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/comstar/lu/stmf_sbd/%.c
709     $(COMPILE.c) -o $@ $<
710     $(CTFCONVERT_O)

712 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/dld/%.c
713     $(COMPILE.c) -o $@ $<
714     $(CTFCONVERT_O)

716 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/dls/%.c
717     $(COMPILE.c) -o $@ $<
718     $(CTFCONVERT_O)

720 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/dmfe/%.c
721     $(COMPILE.c) -o $@ $<

```

```

722     $(CTFCONVERT_O)

724 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/drm/%.c
725     $(COMPILE.c) -o $@ $<
726     $(CTFCONVERT_O)

728 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/efe/%.c
729     $(COMPILE.c) -o $@ $<
730     $(CTFCONVERT_O)

732 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/elx1/%.c
733     $(COMPILE.c) -o $@ $<
734     $(CTFCONVERT_O)

736 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fcoe/%.c
737     $(COMPILE.c) -o $@ $<
738     $(CTFCONVERT_O)

740 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/hme/%.c
741     $(COMPILE.c) -o $@ $<
742     $(CTFCONVERT_O)

744 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/pciex/%.c
745     $(COMPILE.c) -o $@ $<
746     $(CTFCONVERT_O)

748 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/hotplug/hpcsvc/%.c
749     $(COMPILE.c) -o $@ $<
750     $(CTFCONVERT_O)

752 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/pciex/hotplug/%.c
753     $(COMPILE.c) -o $@ $<
754     $(CTFCONVERT_O)

756 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/hotplug/pcihp/%.c
757     $(COMPILE.c) -o $@ $<
758     $(CTFCONVERT_O)

760 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/rds/%.c
761     $(COMPILE.c) -o $@ $<
762     $(CTFCONVERT_O)

764 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/rdsv3/%.c
765     $(COMPILE.c) -o $@ $<
766     $(CTFCONVERT_O)

768 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/iser/%.c
769     $(COMPILE.c) -o $@ $<
770     $(CTFCONVERT_O)

772 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/ibd/%.c
773     $(COMPILE.c) -o $@ $<
774     $(CTFCONVERT_O)

776 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/eoib/%.c
777     $(COMPILE.c) -o $@ $<
778     $(CTFCONVERT_O)

780 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/of/sol_ofs/%.c
781     $(COMPILE.c) -o $@ $<
782     $(CTFCONVERT_O)

784 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/of/sol_ucma/%.c
785     $(COMPILE.c) -o $@ $<
786     $(CTFCONVERT_O)

```

```

788 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/of/sol_umad/%.c
789     $(COMPILE.c) -o $@ $<
790     $(CTFCONVERT_O)

792 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/of/sol_uverbs/%.
793     $(COMPILE.c) -o $@ $<
794     $(CTFCONVERT_O)

796 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/sdp/%.c
797     $(COMPILE.c) -o $@ $<
798     $(CTFCONVERT_O)

800 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/mgt/ibcm/%.c
801     $(COMPILE.c) -o $@ $<
802     $(CTFCONVERT_O)

804 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/mgt/ibdm/%.c
805     $(COMPILE.c) -o $@ $<
806     $(CTFCONVERT_O)

808 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/mgt/ibdma/%.c
809     $(COMPILE.c) -o $@ $<
810     $(CTFCONVERT_O)

812 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/mgt/ibmf/%.c
813     $(COMPILE.c) -o $@ $<
814     $(CTFCONVERT_O)

816 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/ibnex/%.c
817     $(COMPILE.c) -o $@ $<
818     $(CTFCONVERT_O)

820 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/ibt1/%.c
821     $(COMPILE.c) -o $@ $<
822     $(CTFCONVERT_O)

824 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/adapters/tavor/%.c
825     $(COMPILE.c) -o $@ $<
826     $(CTFCONVERT_O)

828 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/adapters/hermon/%.c
829     $(COMPILE.c) -o $@ $<
830     $(CTFCONVERT_O)

832 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ib/clients/daplt/%.c
833     $(COMPILE.c) -o $@ $<
834     $(CTFCONVERT_O)

836 $(OBJSDIR)/%.o: $(COMMONBASE)/iscsi/%.c
837     $(COMPILE.c) -o $@ $<
838     $(CTFCONVERT_O)

840 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/idm/%.c
841     $(COMPILE.c) -o $@ $<
842     $(CTFCONVERT_O)

844 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ipw/%.c
845     $(COMPILE.c) -o $@ $<
846     $(CTFCONVERT_O)

848 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/iwh/%.c
849     $(COMPILE.c) -o $@ $<
850     $(CTFCONVERT_O)

852 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/iwi/%.c
853     $(COMPILE.c) -o $@ $<

```

```

854     $(CTFCONVERT_O)

856 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/iwk/%.c
857     $(COMPILE.c) -o $@ $<
858     $(CTFCONVERT_O)

860 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/iwp/%.c
861     $(COMPILE.c) -o $@ $<
862     $(CTFCONVERT_O)

864 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/kb8042/%.c
865     $(COMPILE.c) -o $@ $<
866     $(CTFCONVERT_O)

868 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/kbtrans/%.c
869     $(COMPILE.c) -o $@ $<
870     $(CTFCONVERT_O)

872 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ksocket/%.c
873     $(COMPILE.c) -o $@ $<
874     $(CTFCONVERT_O)

876 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/aggr/%.c
877     $(COMPILE.c) -o $@ $<
878     $(CTFCONVERT_O)

880 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lp/%.c
881     $(COMPILE.c) -o $@ $<
882     $(CTFCONVERT_O)

884 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/hotspares/%.c
885     $(COMPILE.c) -o $@ $<
886     $(CTFCONVERT_O)

888 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/md/%.c
889     $(COMPILE.c) -o $@ $<
890     $(CTFCONVERT_O)

892 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/mirror/%.c
893     $(COMPILE.c) -o $@ $<
894     $(CTFCONVERT_O)

896 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/notify/%.c
897     $(COMPILE.c) -o $@ $<
898     $(CTFCONVERT_O)

900 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/raid/%.c
901     $(COMPILE.c) -o $@ $<
902     $(CTFCONVERT_O)

904 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/softpart/%.c
905     $(COMPILE.c) -o $@ $<
906     $(CTFCONVERT_O)

908 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/stripe/%.c
909     $(COMPILE.c) -o $@ $<
910     $(CTFCONVERT_O)

912 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/lvm/trans/%.c
913     $(COMPILE.c) -o $@ $<
914     $(CTFCONVERT_O)

916 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mac/%.c
917     $(COMPILE.c) -o $@ $<
918     $(CTFCONVERT_O)

```



```

920 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mac/plugins/%.c
921     $(COMPILE.c) -o $@ $<
922     $(CTFCONVERT_O)

924 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mega_sas/%.c
925     $(COMPILE.c) -o $@ $<
926     $(CTFCONVERT_O)

928 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mii/%.c
929     $(COMPILE.c) -o $@ $<
930     $(CTFCONVERT_O)

932 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mr_sas/%.c
933     $(COMPILE.c) -o $@ $<
934     $(CTFCONVERT_O)

936 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/mpt_sas/%.c
937     $(COMPILE.c) -o $@ $<
938     $(CTFCONVERT_O)

940 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mxfe/%.c
941     $(COMPILE.c) -o $@ $<
942     $(CTFCONVERT_O)

944 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mwl/%.c
945     $(COMPILE.c) -o $@ $<
946     $(CTFCONVERT_O)

948 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/mwl/mwl_fw/%.c
949     $(COMPILE.c) -o $@ $<
950     $(CTFCONVERT_O)

952 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/net80211/%.c
953     $(COMPILE.c) -o $@ $<
954     $(CTFCONVERT_O)

956 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/nge/%.c
957     $(COMPILE.c) -o $@ $<
958     $(CTFCONVERT_O)

960 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/nxge/%.c
961     $(COMPILE.c) -o $@ $<
962     $(CTFCONVERT_O)

964 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/nxge/npi/%.c
965     $(COMPILE.c) -o $@ $<
966     $(CTFCONVERT_O)

968 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/nxge/%.s
969     $(COMPILE.s) -o $@ $<

971 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/pci-ide/%.c
972     $(COMPILE.c) -o $@ $<
973     $(CTFCONVERT_O)

975 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/pcmcia/%.c
976     $(COMPILE.c) -o $@ $<
977     $(CTFCONVERT_O)

979 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/pcan/%.c
980     $(COMPILE.c) -o $@ $<
981     $(CTFCONVERT_O)

983 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/pcn/%.c
984     $(COMPILE.c) -o $@ $<
985     $(CTFCONVERT_O)

```

```

987 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/pcwl/%.c
988     $(COMPILE.c) -o $@ $<
989     $(CTFCONVERT_O)

991 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ppp/sppp/%.c
992     $(COMPILE.c) -o $@ $<
993     $(CTFCONVERT_O)

995 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ppp/spppasyn/%.c
996     $(COMPILE.c) -o $@ $<
997     $(CTFCONVERT_O)

999 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ppp/sppptun/%.c
1000     $(COMPILE.c) -o $@ $<
1001     $(CTFCONVERT_O)

1003 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ral/%.c
1004     $(COMPILE.c) -o $@ $<
1005     $(CTFCONVERT_O)

1007 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/rge/%.c
1008     $(COMPILE.c) -o $@ $<
1009     $(CTFCONVERT_O)

1011 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/rtls/%.c
1012     $(COMPILE.c) -o $@ $<
1013     $(CTFCONVERT_O)

1015 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/rsm/%.c
1016     $(COMPILE.c) -o $@ $<
1017     $(CTFCONVERT_O)

1019 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/rtw/%.c
1020     $(COMPILE.c) -o $@ $<
1021     $(CTFCONVERT_O)

1023 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/rum/%.c
1024     $(COMPILE.c) -o $@ $<
1025     $(CTFCONVERT_O)

1027 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/rwd/%.c
1028     $(COMPILE.c) -o $@ $<
1029     $(CTFCONVERT_O)

1031 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/rwn/%.c
1032     $(COMPILE.c) -o $@ $<
1033     $(CTFCONVERT_O)

1035 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sata/adapters/ahci/%.c
1036     $(COMPILE.c) -o $@ $<
1037     $(CTFCONVERT_O)

1039 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sata/adapters/nv_sata/%.c
1040     $(COMPILE.c) -o $@ $<
1041     $(CTFCONVERT_O)

1043 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sata/adapters/si3124/%.c
1044     $(COMPILE.c) -o $@ $<
1045     $(CTFCONVERT_O)

1047 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sata/impl/%.c
1048     $(COMPILE.c) -o $@ $<
1049     $(CTFCONVERT_O)

1051 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/conf/%.c

```

```

1052 $(COMPILE.c) -o $@ $<
1053 $(CTFCONVERT_O)

1055 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/impl/%.c
1056 $(COMPILE.c) -o $@ $<
1057 $(CTFCONVERT_O)

1059 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/targets/%.c
1060 $(COMPILE.c) -o $@ $<
1061 $(CTFCONVERT_O)

1063 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/%.c
1064 $(COMPILE.c) -o $@ $<
1065 $(CTFCONVERT_O)

1067 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/blk2scsa/%.c
1068 $(COMPILE.c) -o $@ $<
1069 $(CTFCONVERT_O)

1071 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/%.c
1072 $(COMPILE.c) -o $@ $<
1073 $(CTFCONVERT_O)

1075 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/fop
1076 $(COMPILE.c) -o $@ $<
1077 $(CTFCONVERT_O)

1079 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/ulp/%.c
1080 $(COMPILE.c) -o $@ $<
1081 $(CTFCONVERT_O)

1083 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/impl/%.c
1084 $(COMPILE.c) -o $@ $<
1085 $(CTFCONVERT_O)

1087 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/fca/qlc/%.c
1088 $(COMPILE.c) -o $@ $<
1089 $(CTFCONVERT_O)

1091 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/fca/qlge/%.c
1092 $(COMPILE.c) -o $@ $<
1093 $(CTFCONVERT_O)

1095 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/fca/emlxs/%.c
1096 $(COMPILE.c) -o $@ $<
1097 $(CTFCONVERT_O)

1099 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/fca/oce/%.c
1100 $(COMPILE.c) -o $@ $<
1101 $(CTFCONVERT_O)

1103 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/fca/fcoei/%.c
1104 $(COMPILE.c) -o $@ $<
1105 $(CTFCONVERT_O)

1107 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sdcard/adapters/sdhost/%.c
1108 $(COMPILE.c) -o $@ $<
1109 $(CTFCONVERT_O)

1111 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sdcard/impl/%.c
1112 $(COMPILE.c) -o $@ $<
1113 $(CTFCONVERT_O)

1115 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sdcard/targets/sdcard/%.c
1116 $(COMPILE.c) -o $@ $<
1117 $(CTFCONVERT_O)

```

```

1119 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/sfe/%.c
1120 $(COMPILE.c) -o $@ $<
1121 $(CTFCONVERT_O)

1123 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/simnet/%.c
1124 $(COMPILE.c) -o $@ $<
1125 $(CTFCONVERT_O)

1127 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/softmac/%.c
1128 $(COMPILE.c) -o $@ $<
1129 $(CTFCONVERT_O)

1131 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/uath/%.c
1132 $(COMPILE.c) -o $@ $<
1133 $(CTFCONVERT_O)

1135 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/uath/uath_fw/%.c
1136 $(COMPILE.c) -o $@ $<
1137 $(CTFCONVERT_O)

1139 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ural/%.c
1140 $(COMPILE.c) -o $@ $<
1141 $(CTFCONVERT_O)

1143 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/urtw/%.c
1144 $(COMPILE.c) -o $@ $<
1145 $(CTFCONVERT_O)

1147 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/audio/usb_ac/%.
1148 $(COMPILE.c) -o $@ $<
1149 $(CTFCONVERT_O)

1151 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/audio/usb_as/%.
1152 $(COMPILE.c) -o $@ $<
1153 $(CTFCONVERT_O)

1155 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/audio/usb_ah/%.
1156 $(COMPILE.c) -o $@ $<
1157 $(CTFCONVERT_O)

1159 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbskel/%.c
1160 $(COMPILE.c) -o $@ $<
1161 $(CTFCONVERT_O)

1163 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/video/usbvc/%.c
1164 $(COMPILE.c) -o $@ $<
1165 $(CTFCONVERT_O)

1167 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/hwarc/%.c
1168 $(COMPILE.c) -o $@ $<
1169 $(CTFCONVERT_O)

1171 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/hid/%.c
1172 $(COMPILE.c) -o $@ $<
1173 $(CTFCONVERT_O)

1175 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/hidparser/%.c
1176 $(COMPILE.c) -o $@ $<
1177 $(CTFCONVERT_O)

1179 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/printer/%.c
1180 $(COMPILE.c) -o $@ $<
1181 $(CTFCONVERT_O)

1183 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbkbm/%.c

```

```

1184 $(COMPILE.c) -o $@ $<
1185 $(CTFCONVERT_O)

1187 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbms/%.c
1188 $(COMPILE.c) -o $@ $<
1189 $(CTFCONVERT_O)

1191 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbinput/usbwcm
1192 $(COMPILE.c) -o $@ $<
1193 $(CTFCONVERT_O)

1195 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/ugen/%.c
1196 $(COMPILE.c) -o $@ $<
1197 $(CTFCONVERT_O)

1199 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbser/%.c
1200 $(COMPILE.c) -o $@ $<
1201 $(CTFCONVERT_O)

1203 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbser/usbsacm/
1204 $(COMPILE.c) -o $@ $<
1205 $(CTFCONVERT_O)

1207 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbser/usbftdi/
1208 $(COMPILE.c) -o $@ $<
1209 $(CTFCONVERT_O)

1211 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbser/usbser_k
1212 $(COMPILE.c) -o $@ $<
1213 $(CTFCONVERT_O)

1215 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbser/usbsprl/
1216 $(COMPILE.c) -o $@ $<
1217 $(CTFCONVERT_O)

1219 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/wusb_df/%.c
1220 $(COMPILE.c) -o $@ $<
1221 $(CTFCONVERT_O)

1223 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/hwa1480_fw/%.c
1224 $(COMPILE.c) -o $@ $<
1225 $(CTFCONVERT_O)

1227 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/wusb_ca/%.c
1228 $(COMPILE.c) -o $@ $<
1229 $(CTFCONVERT_O)

1231 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/clients/usbecm/%.c
1232 $(COMPILE.c) -o $@ $<
1233 $(CTFCONVERT_O)

1235 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/hcd/openhci/%.c
1236 $(COMPILE.c) -o $@ $<
1237 $(CTFCONVERT_O)

1239 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/hcd/ehci/%.c
1240 $(COMPILE.c) -o $@ $<
1241 $(CTFCONVERT_O)

1243 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/hcd/uhci/%.c
1244 $(COMPILE.c) -I.../common -o $@ $<
1245 $(CTFCONVERT_O)

1247 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/hubd/%.c
1248 $(COMPILE.c) -o $@ $<
1249 $(CTFCONVERT_O)

```

```

1251 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/scsa2usb/%.c
1252 $(COMPILE.c) -o $@ $<
1253 $(CTFCONVERT_O)

1255 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/usb_mid/%.c
1256 $(COMPILE.c) -o $@ $<
1257 $(CTFCONVERT_O)

1259 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/usb_ia/%.c
1260 $(COMPILE.c) -o $@ $<
1261 $(CTFCONVERT_O)

1263 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/usba/%.c
1264 $(COMPILE.c) -o $@ $<
1265 $(CTFCONVERT_O)

1267 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/usba10/%.c
1268 $(COMPILE.c) -o $@ $<
1269 $(CTFCONVERT_O)

1271 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/usb/hwa/hwahc/%.c
1272 $(COMPILE.c) -o $@ $<
1273 $(CTFCONVERT_O)

1275 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/uwb/uwba/%.c
1276 $(COMPILE.c) -o $@ $<
1277 $(CTFCONVERT_O)

1279 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/vuidmice/%.c
1280 $(COMPILE.c) -o $@ $<
1281 $(CTFCONVERT_O)

1283 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/vnic/%.c
1284 $(COMPILE.c) -o $@ $<
1285 $(CTFCONVERT_O)

1287 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/wpi/%.c
1288 $(COMPILE.c) -o $@ $<
1289 $(CTFCONVERT_O)

1291 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/zyd/%.c
1292 $(COMPILE.c) -o $@ $<
1293 $(CTFCONVERT_O)

1295 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/chxge/com/%.c
1296 $(COMPILE.c) -o $@ $<
1297 $(CTFCONVERT_O)

1299 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/chxge/%.c
1300 $(COMPILE.c) -o $@ $<
1301 $(CTFCONVERT_O)

1303 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ixgb/%.c
1304 $(COMPILE.c) -o $@ $<
1305 $(CTFCONVERT_O)

1307 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/xge/drv/%.c
1308 $(COMPILE.c) -o $@ $<
1309 $(CTFCONVERT_O)

1311 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/xge/hal/xgehal/%.c
1312 $(COMPILE.c) -o $@ $<
1313 $(CTFCONVERT_O)

1315 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/e1000g/%.c

```

```

1316 $(COMPILE.c) -o $@ $<
1317 $(CTFCONVERT_O)

1319 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/igb/%.c
1320 $(COMPILE.c) -o $@ $<
1321 $(CTFCONVERT_O)

1323 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/iprb/%.c
1324 $(COMPILE.c) -o $@ $<
1325 $(CTFCONVERT_O)

1327 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ixgbe/%.c
1328 $(COMPILE.c) -o $@ $<
1329 $(CTFCONVERT_O)

1331 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/ntxn/%.c
1332 $(COMPILE.c) -o $@ $<
1333 $(CTFCONVERT_O)

1335 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/myri10ge/drv/%.c
1336 $(COMPILE.c) -o $@ $<
1337 $(CTFCONVERT_O)

1339 $(OBJSDIR)/%.o: $(UTSBASE)/common/ipp/%.c
1340 $(COMPILE.c) -o $@ $<
1341 $(CTFCONVERT_O)

1343 $(OBJSDIR)/%.o: $(UTSBASE)/common/ipp/ipgpc/%.c
1344 $(COMPILE.c) -o $@ $<
1345 $(CTFCONVERT_O)

1347 $(OBJSDIR)/%.o: $(UTSBASE)/common/ipp/dlcosmk/%.c
1348 $(COMPILE.c) -o $@ $<
1349 $(CTFCONVERT_O)

1351 $(OBJSDIR)/%.o: $(UTSBASE)/common/ipp/flowacct/%.c
1352 $(COMPILE.c) -o $@ $<
1353 $(CTFCONVERT_O)

1355 $(OBJSDIR)/%.o: $(UTSBASE)/common/ipp/dscpmk/%.c
1356 $(COMPILE.c) -o $@ $<
1357 $(CTFCONVERT_O)

1359 $(OBJSDIR)/%.o: $(UTSBASE)/common/ipp/meters/%.c
1360 $(COMPILE.c) -o $@ $<
1361 $(CTFCONVERT_O)

1363 $(OBJSDIR)/%.o: $(UTSBASE)/common/kiconv/kiconv_emea/%.c
1364 $(COMPILE.c) -o $@ $<
1365 $(CTFCONVERT_O)

1367 $(OBJSDIR)/%.o: $(UTSBASE)/common/kiconv/kiconv_ja/%.c
1368 $(COMPILE.c) -o $@ $<
1369 $(CTFCONVERT_O)

1371 $(OBJSDIR)/%.o: $(UTSBASE)/common/kiconv/kiconv_ko/%.c
1372 $(COMPILE.c) -o $@ $<
1373 $(CTFCONVERT_O)

1375 $(OBJSDIR)/%.o: $(UTSBASE)/common/kiconv/kiconv_sc/%.c
1376 $(COMPILE.c) -o $@ $<
1377 $(CTFCONVERT_O)

1379 $(OBJSDIR)/%.o: $(UTSBASE)/common/kiconv/kiconv_tc/%.c
1380 $(COMPILE.c) -o $@ $<
1381 $(CTFCONVERT_O)

```

```

1383 $(OBJSDIR)/%.o: $(UTSBASE)/common/kmdb/%.c
1384 $(COMPILE.c) -o $@ $<
1385 $(CTFCONVERT_O)

1387 $(OBJSDIR)/%.o: $(UTSBASE)/common/ktli/%.c
1388 $(COMPILE.c) -o $@ $<
1389 $(CTFCONVERT_O)

1391 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/iscsi/%.c
1392 $(COMPILE.c) -o $@ $<
1393 $(CTFCONVERT_O)

1395 $(OBJSDIR)/%.o: $(COMMONBASE)/iscsi/%.c
1396 $(COMPILE.c) -o $@ $<
1397 $(CTFCONVERT_O)

1399 $(OBJSDIR)/%.o: $(UTSBASE)/common/inet/kifconf/%.c
1400 $(COMPILE.c) -o $@ $<
1401 $(CTFCONVERT_O)

1403 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/vr/%.c
1404 $(COMPILE.c) -o $@ $<
1405 $(CTFCONVERT_O)

1407 $(OBJSDIR)/%.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 $(OBJSDIR)/bootrd.o := CPPFLAGS += -DKOBJ_OVERRIDES
1415 $(OBJSDIR)/doreloc.o := CPPFLAGS += -DKOBJ_OVERRIDES
1416 $(OBJSDIR)/kobj.o := CPPFLAGS += -DKOBJ_OVERRIDES
1417 $(OBJSDIR)/kobj_boot.o := CPPFLAGS += -DKOBJ_OVERRIDES
1418 $(OBJSDIR)/kobj_bootflags.o := CPPFLAGS += -DKOBJ_OVERRIDES
1419 $(OBJSDIR)/kobj_convrelstr.o := CPPFLAGS += -DKOBJ_OVERRIDES
1420 $(OBJSDIR)/kobj_isa.o := CPPFLAGS += -DKOBJ_OVERRIDES
1421 $(OBJSDIR)/kobj_kdi.o := CPPFLAGS += -DKOBJ_OVERRIDES
1422 $(OBJSDIR)/kobj_lm.o := CPPFLAGS += -DKOBJ_OVERRIDES
1423 $(OBJSDIR)/kobj_reloc.o := CPPFLAGS += -DKOBJ_OVERRIDES
1424 $(OBJSDIR)/kobj_stubs.o := CPPFLAGS += -DKOBJ_OVERRIDES
1425 $(OBJSDIR)/kobj_subr.o := CPPFLAGS += -DKOBJ_OVERRIDES

1427 $(OBJSDIR)/%.o: $(UTSBASE)/common/krtld/%.c
1428 $(COMPILE.c) -o $@ $<
1429 $(CTFCONVERT_O)

1431 $(OBJSDIR)/%.o: $(COMMONBASE)/list/%.c
1432 $(COMPILE.c) -o $@ $<
1433 $(CTFCONVERT_O)

1435 $(OBJSDIR)/%.o: $(COMMONBASE)/lvm/%.c
1436 $(COMPILE.c) -o $@ $<
1437 $(CTFCONVERT_O)

1439 $(OBJSDIR)/%.o: $(COMMONBASE)/lzma/%.c
1440 $(COMPILE.c) -o $@ $<
1441 $(CTFCONVERT_O)

1443 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/md4/%.c
1444 $(COMPILE.c) -o $@ $<
1445 $(CTFCONVERT_O)

1447 $(OBJSDIR)/%.o: $(COMMONBASE)/crypto/md5/%.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:      $(UTSBASE)/common/fs/cacheefs/%.c
1721      @$(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:      $(UTSBASE)/common/fs/dcfis/%.c
1730      @$(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:      $(UTSBASE)/common/fs/smbclnt/net smb/%.c
1784      @$(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))

```



```

1976 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiocmi/%.c
1977 @$(LHEAD) $(LINT.c) $< $(LTAIL))

1979 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiocmihd/%.c
1980 @$(LHEAD) $(LINT.c) $< $(LTAIL))

1982 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audioens/%.c
1983 @$(LHEAD) $(LINT.c) $< $(LTAIL))

1985 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audioemu10k/%.c
1986 @$(LHEAD) $(LINT.c) $< $(LTAIL))

1988 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiohd/%.c
1989 @$(LHEAD) $(LINT.c) $< $(LTAIL))

1991 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audioixp/%.c
1992 @$(LHEAD) $(LINT.c) $< $(LTAIL))

1994 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiols/%.c
1995 @$(LHEAD) $(LINT.c) $< $(LTAIL))

1997 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiopci/%.c
1998 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2000 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiop16x/%.c
2001 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2003 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiosolo/%.c
2004 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2006 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiots/%.c
2007 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2009 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiovia823x/%.c
2010 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2012 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiovia97/%.c
2013 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2015 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/bfe/%.c
2016 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2018 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/bpf/%.c
2019 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2021 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/bge/%.c
2022 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2024 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/blkdev/%.c
2025 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2027 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/cardbus/%.c
2028 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2030 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/lu/stmf_sbd/%.c
2031 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2033 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/port/fct/%.c
2034 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2036 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/port/qlt/%.c
2037 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2039 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/port/srpt/%.c
2040 @$(LHEAD) $(LINT.c) $< $(LTAIL))

```

```

2042 $(LINTSDIR)/%.ln: $(COMMONBASE)/iscsit/%.c
2043 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2045 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/port/fcoet/%.c
2046 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2048 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/port/iscsit/%.c
2049 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2051 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/port/pppt/%.c
2052 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2054 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/comstar/stmf/%.c
2055 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2057 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/dld/%.c
2058 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2060 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/dls/%.c
2061 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2063 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/dmfe/%.c
2064 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2066 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/drm/%.c
2067 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2069 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/efe/%.c
2070 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2072 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/elx1/%.c
2073 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2075 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fcoe/%.c
2076 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2078 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/hme/%.c
2079 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2081 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/pciex/%.c
2082 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2084 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/hotplug/hpcsvc/%.c
2085 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2087 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/pciex/hotplug/%.c
2088 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2090 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/hotplug/pcihp/%.c
2091 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2093 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/rds/%.c
2094 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2096 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/rds3/%.c
2097 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2099 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/iser/%.c
2100 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2102 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/ibd/%.c
2103 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2105 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/eoib/%.c
2106 @$(LHEAD) $(LINT.c) $< $(LTAIL))

```

```

2108 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/of/sol_ofs/%.c
2109     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2111 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/of/sol_ucma/%.c
2112     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2114 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/of/sol_umad/%.c
2115     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2117 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/of/sol_uverbs/%.
2118     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2120 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/sdp/%.c
2121     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2123 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/mgt/ibcm/%.c
2124     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2126 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/mgt/ibdm/%.c
2127     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2129 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/mgt/ibdma/%.c
2130     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2132 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/mgt/ibmf/%.c
2133     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2135 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/ibnex/%.c
2136     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2138 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/ibt1/%.c
2139     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2141 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/adapters/tavor/%.c
2142     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2144 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/adapters/hermon/%.c
2145     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2147 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ib/clients/daplt/%.c
2148     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2150 $(LINTSDIR)/%.ln: $(COMMONBASE)/iscsi/%.c
2151     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2153 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/idm/%.c
2154     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2156 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ipw/%.c
2157     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2159 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/iwh/%.c
2160     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2162 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/iwi/%.c
2163     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2165 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/iwk/%.c
2166     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2168 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/iwp/%.c
2169     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2171 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/kb8042/%.c
2172     @$(LHEAD) $(LINT.c) $< $(LTAIL))

```

```

2174 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/kbtrans/%.c
2175     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2177 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ksocket/%.c
2178     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2180 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/aggr/%.c
2181     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2183 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lp/%.c
2184     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2186 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/hotspares/%.c
2187     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2189 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/md/%.c
2190     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2192 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/mirror/%.c
2193     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2195 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/raid/%.c
2196     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2198 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/softpart/%.c
2199     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2201 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/stripes/%.c
2202     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2204 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/notify/%.c
2205     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2207 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/lvm/trans/%.c
2208     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2210 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mac/%.c
2211     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2213 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mac/plugins/%.c
2214     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2216 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mega_sas/%.c
2217     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2219 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mii/%.c
2220     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2222 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mr_sas/%.c
2223     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2225 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/adapters/mpt_sas/%.c
2226     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2228 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mxfe/%.c
2229     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2231 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mwl/%.c
2232     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2234 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/mwl/mwl_fw/%.c
2235     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2237 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/net80211/%.c
2238     @$(LHEAD) $(LINT.c) $< $(LTAIL))

```

```

2240 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/nge/%.c
2241     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2243 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/nxge/%.c
2244     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2246 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/nxge/%.s
2247     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2249 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/nxge/npi/%.c
2250     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2252 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/pci-ide/%.c
2253     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2255 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/pcmcia/%.c
2256     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2258 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/pcan/%.c
2259     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2261 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/pcn/%.c
2262     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2264 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/pcwl/%.c
2265     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2267 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ppp/sppp/%.c
2268     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2270 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ppp/sppasyn/%.c
2271     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2273 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ppp/sppptun/%.c
2274     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2276 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ral/%.c
2277     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2279 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/rge/%.c
2280     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2282 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/rtls/%.c
2283     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2285 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/rsm/%.c
2286     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2288 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/rtw/%.c
2289     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2291 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/rum/%.c
2292     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2294 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/rwd/%.c
2295     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2297 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/rwn/%.c
2298     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2300 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sata/adapters/ahci/%.c
2301     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2303 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sata/adapters/nv_sata/%.c
2304     @$(LHEAD) $(LINT.c) $< $(LTAIL)

```

```

2306 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sata/adapters/si3124/%.c
2307     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2309 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sata/impl/%.c
2310     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2312 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/adapters/%.c
2313     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2315 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/adapters/blk2scsa/%.c
2316     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2318 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/adapters/pmcs/%.c
2319     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2321 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/%.c
2322     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2324 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/adapters/scsi_vhci/fop
2325     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2327 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fibre-channel/ulp/%.c
2328     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2330 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fibre-channel/impl/%.c
2331     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2333 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fibre-channel/fca/qlc/%.c
2334     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2336 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fibre-channel/fca/qlge/%.c
2337     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2339 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fibre-channel/fca/emlxs/%.c
2340     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2342 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fibre-channel/fca/oce/%.c
2343     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2345 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/fibre-channel/fca/fcoei/%.c
2346     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2348 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/conf/%.c
2349     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2351 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/impl/%.c
2352     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2354 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/targets/%.c
2355     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2357 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sdcard/adapters/sdhost/%.c
2358     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2360 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sdcard/impl/%.c
2361     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2363 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sdcard/targets/sdcard/%.c
2364     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2366 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/sfe/%.c
2367     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2369 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/simnet/%.c
2370     @$(LHEAD) $(LINT.c) $< $(LTAIL)

```

```

2372 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/softmac/%.c
2373     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2375 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/uath/%.c
2376     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2378 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/uath/uath_fw/%.c
2379     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2381 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ural/%.c
2382     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2384 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/urtw/%.c
2385     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2387 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/audio/usb_ac/%.
2388     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2390 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/audio/usb_as/%.
2391     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2393 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/audio/usb_ah/%.
2394     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2396 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbskel/%.c
2397     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2399 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/video/usbvc/%.c
2400     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2402 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/hwarc/%.c
2403     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2405 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/hid/%.c
2406     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2408 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/hidparser/%.c
2409     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2411 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbkbm/%.c
2412     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2414 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbms/%.c
2415     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2417 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbinput/usbwcm
2418     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2420 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/ugen/%.c
2421     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2423 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/printer/%.c
2424     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2426 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbser/%.c
2427     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2429 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbser/usbsacm/
2430     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2432 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbser/usbftdi/
2433     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2435 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbser/usbser_k
2436     @$(LHEAD) $(LINT.c) $< $(LTAIL))

```

```

2438 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbser/usbsprl/
2439     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2441 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/wusb_df/%.c
2442     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2444 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/hwal480_fw/%.c
2445     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2447 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/wusb_ca/%.c
2448     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2450 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbecm/%.c
2451     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2453 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/hcd/openhci/%.c
2454     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2456 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/hcd/ehci/%.c
2457     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2459 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/hcd/uhci/%.c
2460     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2462 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/hubd/%.c
2463     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2465 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/scsa2usb/%.c
2466     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2468 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/usb_mid/%.c
2469     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2471 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/usb_ia/%.c
2472     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2474 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/usba/%.c
2475     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2477 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/usba10/%.c
2478     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2480 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/uwb/uwba/%.c
2481     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2483 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/hwa/hwahc/%.c
2484     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2486 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/vuidmice/%.c
2487     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2489 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/vnic/%.c
2490     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2492 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/wpi/%.c
2493     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2495 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/zyd/%.c
2496     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2498 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/chxge/com/%.c
2499     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2501 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/chxge/%.c
2502     @$(LHEAD) $(LINT.c) $< $(LTAIL))

```

```

2504 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ixgb/%.c
2505     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2507 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/xge/drv/%.c
2508     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2510 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/xge/hal/xgehal/%.c
2511     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2513 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/e1000g/%.c
2514     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2516 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/igb/%.c
2517     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2519 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/iprb/%.c
2520     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2522 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ixgbe/%.c
2523     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2525 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/ntxn/%.c
2526     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2528 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/myril0ge/drv/%.c
2529     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2531 $(LINTSDIR)/%.ln: $(UTSBASE)/common/ipp/%.c
2532     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2534 $(LINTSDIR)/%.ln: $(UTSBASE)/common/ipp/ipgpc/%.c
2535     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2537 $(LINTSDIR)/%.ln: $(UTSBASE)/common/ipp/dlcosmk/%.c
2538     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2540 $(LINTSDIR)/%.ln: $(UTSBASE)/common/ipp/flowacct/%.c
2541     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2543 $(LINTSDIR)/%.ln: $(UTSBASE)/common/ipp/dscpmk/%.c
2544     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2546 $(LINTSDIR)/%.ln: $(UTSBASE)/common/ipp/meters/%.c
2547     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2549 $(LINTSDIR)/%.ln: $(UTSBASE)/common/kiconv/kiconv_emea/%.c
2550     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2552 $(LINTSDIR)/%.ln: $(UTSBASE)/common/kiconv/kiconv_ja/%.c
2553     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2555 $(LINTSDIR)/%.ln: $(UTSBASE)/common/kiconv/kiconv_ko/%.c
2556     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2558 $(LINTSDIR)/%.ln: $(UTSBASE)/common/kiconv/kiconv_sc/%.c
2559     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2561 $(LINTSDIR)/%.ln: $(UTSBASE)/common/kiconv/kiconv_tc/%.c
2562     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2564 $(LINTSDIR)/%.ln: $(UTSBASE)/common/kmdb/%.c
2565     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2567 $(LINTSDIR)/%.ln: $(UTSBASE)/common/krtld/%.c
2568     @$(LHEAD) $(LINT.c) $< $(LTAIL)

```

```

2570 $(LINTSDIR)/%.ln: $(UTSBASE)/common/ktli/%.c
2571     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2573 $(LINTSDIR)/%.ln: $(COMMONBASE)/list/%.c
2574     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2576 $(LINTSDIR)/%.ln: $(COMMONBASE)/lvm/%.c
2577     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2579 $(LINTSDIR)/%.ln: $(COMMONBASE)/lzma/%.c
2580     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2582 $(LINTSDIR)/%.ln: $(COMMONBASE)/crypto/md4/%.c
2583     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2585 $(LINTSDIR)/%.ln: $(COMMONBASE)/crypto/md5/%.c
2586     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2588 $(LINTSDIR)/%.ln: $(COMMONBASE)/net/dhcp/%.c
2589     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2591 $(LINTSDIR)/%.ln: $(COMMONBASE)/nvpair/%.c
2592     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2594 $(LINTSDIR)/%.ln: $(UTSBASE)/common/os/%.c
2595     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2597 $(LINTSDIR)/%.ln: $(UTSBASE)/common/rpc/%.c
2598     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2600 $(LINTSDIR)/%.ln: $(UTSBASE)/common/pcmcia/cs/%.c
2601     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2603 $(LINTSDIR)/%.ln: $(UTSBASE)/common/pcmcia/cis/%.c
2604     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2606 $(LINTSDIR)/%.ln: $(UTSBASE)/common/pcmcia/nexus/%.c
2607     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2609 $(LINTSDIR)/%.ln: $(UTSBASE)/common/pcmcia/pcs/%.c
2610     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2612 $(LINTSDIR)/%.ln: $(UTSBASE)/common/rpc/%.c
2613     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2615 $(LINTSDIR)/%.ln: $(UTSBASE)/common/rpc/sec/%.c
2616     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2618 $(LINTSDIR)/%.ln: $(UTSBASE)/common/rpc/sec_gss/%.c
2619     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2621 $(LINTSDIR)/%.ln: $(COMMONBASE)/crypto/shal/%.c
2622     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2624 $(LINTSDIR)/%.ln: $(COMMONBASE)/crypto/sha2/%.c
2625     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2627 $(LINTSDIR)/%.ln: $(UTSBASE)/common/syscall/%.c
2628     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2630 $(LINTSDIR)/%.ln: $(UTSBASE)/common/tnf/%.c
2631     @$(LHEAD) $(LINT.c) $< $(LTAIL)

2633 $(LINTSDIR)/%.ln: $(COMMONBASE)/tsol/%.c
2634     @$(LHEAD) $(LINT.c) $< $(LTAIL)

```

```
2636 $(LINTSDIR)/%.ln: $(COMMONBASE)/util/%.c
2637 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2639 $(LINTSDIR)/%.ln: $(COMMONBASE)/unicode/%.c
2640 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2642 $(LINTSDIR)/%.ln: $(UTSBASE)/common/vm/%.c
2643 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2645 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/scsi/adapters/iscsi/%.c
2646 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2648 $(LINTSDIR)/%.ln: $(COMMONBASE)/iscsi/%.c
2649 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2651 $(LINTSDIR)/%.ln: $(UTSBASE)/common/inet/kifconf/%.c
2652 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2654 ZMODLINTFLAGS = -erroff=E_CONSTANT_CONDITION

2656 $(LINTSDIR)/%.ln: $(UTSBASE)/common/zmod/%.c
2657 @$(LHEAD) $(LINT.c) $(ZMODLINTFLAGS) $< $(LTAIL))

2659 $(LINTSDIR)/zlib_obj.ln: $(ZLIB_OBJS:%.o=$(LINTSDIR)/%.ln) \
2660 $(UTSBASE)/common/zmod/zlib_lint.c
2661 @$(LHEAD) $(LINT.c) -C $(LINTSDIR)/zlib_obj \
2662 $(UTSBASE)/common/zmod/zlib_lint.c $(LTAIL))

2664 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/hxge/%.c
2665 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2667 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/tpm/%.c
2668 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2670 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/tpm/%.s
2671 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2673 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/vr/%.c
2674 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2676 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/yge/%.c
2677 @$(LHEAD) $(LINT.c) $< $(LTAIL))

2679 $(LINTSDIR)/%.ln: $(COMMONBASE)/fsreparse/%.c
2680 @$(LHEAD) $(LINT.c) $< $(LTAIL))
```

```

*****
19475 Wed Jun 13 12:04:29 2012
new/usr/src/uts/common/inet/dccp/dccp.c
%B
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
24  * Copyright 2012 David Hoepfner. All rights reserved.
25  */

27 /*
28  * This file implements the Data Congestion Control Protocol (DCCP).
29  */

31 #include <sys/types.h>
32 #include <sys/stream.h>
33 #include <sys/stropts.h>
34 #include <sys/strlog.h>
35 #include <sys/strsun.h>
36 #define _SUN_TPI_VERSION 2
37 #include <sys/tihdr.h>
38 #include <sys/socket.h>
39 #include <sys/socketvar.h>
40 #include <sys/sockio.h>
41 #include <sys/vtrace.h>
42 #include <sys/sdt.h>
43 #include <sys/debug.h>
44 #include <sys/isa_defs.h>
45 #include <sys/tsol/label.h>
46 #include <sys/tsol/tnet.h>
47 #include <inet/kstatcom.h>
48 #include <inet/snmpcom.h>

50 #include <sys/cmn_err.h>

52 #include "dccp_impl.h"
53 #include "dccp_stack.h"

55 /* Setable in /etc/system */
56 uint_t dccp_bind_fanout_size = DCCP_BIND_FANOUT_SIZE;

58 static void    dccp_notify(void *, ip_xmit_attr_t *, ixa_notify_type_t,
59                ixa_notify_arg_t);

61 /* Functions to register netstack */

```

```

62 static void    *dccp_stack_init(netstackid_t, netstack_t *);
63 static void    dccp_stack_fini(netstackid_t, void *);

65 static int     dccp_openv4(queue_t *, dev_t *, int, int, cred_t *);
66 static int     dccp_openv6(queue_t *, dev_t *, int, int, cred_t *);

68 /* Write service routine */
69 static void    dccp_wsrv(queue_t *);

71 /* Connection related functions */
72 static int     dccp_connect_ipv4(dccp_t *, ipaddr_t *, in_port_t, uint_t);
73 static int     dccp_connect_ipv6(dccp_t *, in6_addr_t *, in_port_t, uint32_t,
74                uint_t, uint32_t);

76 struct module_info dccp_rinfo = {
77     DCCP_MOD_ID, DCCP_MOD_NAME, 0, INFPSZ, DCCP_RECV_HIWATER,
78     DCCP_RECV_LOWATER
79 };

81 static struct module_info dccp_winfo = {
82     DCCP_MOD_ID, DCCP_MOD_NAME, 0, INFPSZ, 127, 16
83 };

85 /*
86  * Queue information structure with DCCP entry points.
87  */
88 struct qinit dccp_rinitv4 = {
89     NULL, (pfi_t)dccp_rsrv, dccp_openv4, dccp_tpi_close, NULL, &dccp_rinfo
90 };

92 struct qinit dccp_rinitv6 = {
93     NULL, (pfi_t)dccp_rsrv, dccp_openv6, dccp_tpi_close, NULL, &dccp_rinfo
94 };

96 struct qinit dccp_winit = {
97     (pfi_t)dccp_wput, (pfi_t)dccp_wsrv, NULL, NULL, NULL, &dccp_winfo
98 };

100 /* AF_INET /dev/dccp */
101 struct streamtab dccpinfov4 = {
102     &dccp_rinitv4, &dccp_winit
103 };

105 /* AF_INET6 /dev/dccp6 */
106 struct streamtab dccpinfov6 = {
107     &dccp_rinitv6, &dccp_winit
108 };

110 /*
111  * Tunables.
112  */
113 extern mod_prop_info_t dccp_propinfo_tbl[];
114 extern int dccp_propinfo_count;

116 /*
117  * Register DCCP netstack.
118  */
119 void
120 dccp_ddi_g_init(void)
121 {
122     netstack_register(NS_DCCP, dccp_stack_init, NULL, dccp_stack_fini);
123 }

125 #define INET_NAME    "ip"

127 /*

```

```

128 * Initialize the DCCP stack instance.
129 */
130 static void *
131 dccp_stack_init(netstackid_t stackid, netstack_t *ns)
132 {
133     dccp_stack_t    *dccps;
134     major_t         major;
135     size_t          arrsz;
136     int             error;
137     int             i;

139     dccps = kmem_zalloc(sizeof (*dccps), KM_SLEEP);
140     if (dccps == NULL) {
141         return (NULL);
142     }
143     dccps->dccps_netstack = ns;

145     /* Ports */
146     mutex_init(&dccps->dccps_epriv_port_lock, NULL, MUTEX_DEFAULT, NULL);
147     dccps->dccps_num_epriv_ports = DCCP_NUM_EPRIV_PORTS;
148     dccps->dccps_epriv_ports[0] = ULP_DEF_EPRIV_PORT1;
149     dccps->dccps_epriv_ports[1] = ULP_DEF_EPRIV_PORT2;
150     dccps->dccps_min_anonpriv_port = 512;

152     dccps->dccps_bind_fanout_size = dccp_bind_fanout_size;

154     /* Bind fanout */
155     dccps->dccps_bind_fanout = kmem_zalloc(dccps->dccps_bind_fanout_size *
156     sizeof (dccp_df_t), KM_SLEEP);
157     for (i = 0; i < dccps->dccps_bind_fanout_size; i++) {
158         mutex_init(&dccps->dccps_bind_fanout[i].df_lock, NULL,
159         MUTEX_DEFAULT, NULL);
160     }

162     /* Tunable properties */
163     arrsz = dccp_propinfo_count * sizeof (mod_prop_info_t);
164     dccps->dccps_propinfo_tbl = kmem_alloc(arrsz, KM_SLEEP);
165     if (dccps->dccps_propinfo_tbl == NULL) {
166         kmem_free(dccps, sizeof (*dccps));
167         return (NULL);
168     }
169     bcopy(dccp_propinfo_tbl, dccps->dccps_propinfo_tbl, arrsz);

171     /* Allocate per netstack cpu stats */
172     mutex_enter(&cpu_lock);
173     dccps->dccps_sc_cnt = MAX(ncpus, boot_ncpus);
174     mutex_exit(&cpu_lock);

176     dccps->dccps_sc = kmem_zalloc(max_ncpus * sizeof (dccp_stats_cpu_t *),
177     KM_SLEEP);
178     for (i = 0; i < dccps->dccps_sc_cnt; i++) {
179         dccps->dccps_sc[i] = kmem_zalloc(sizeof (dccp_stats_cpu_t),
180         KM_SLEEP);
181     }

183     /* Driver major number */
184     major = mod_name_to_major(INET_NAME);
185     error = ldi_ident_from_major(major, &dccps->dccps_ldi_ident);
186     ASSERT(error == 0);

188     return (dccps);
189 }

191 /*
192 * Destroy the DCCP stack instance.
193 */

```

```

194 void
195 dccp_ddi_g_destroy(void)
196 {
197     cmn_err(CE_NOTE, "dccp.c: dccp_ddi_g_destroy\n");

199     netstack_unregister(NS_DCCP);
200 }

202 static void
203 dccp_stack_fini(netstackid_t stackid, void *arg)
204 {
205     dccp_stack_t    *dccps = (dccp_stack_t *)arg;
206     int             i;

208     /* Cpu stats */
209     for (i = 0; i < dccps->dccps_sc_cnt; i++) {
210         kmem_free(dccps->dccps_sc[i], sizeof (dccp_stats_cpu_t));
211     }
212     kmem_free(dccps->dccps_sc, max_ncpus * sizeof (dccp_stats_cpu_t *));

214     /* Tunable properties */
215     kmem_free(dccps->dccps_propinfo_tbl,
216     dccp_propinfo_count * sizeof (mod_prop_info_t));
217     dccps->dccps_propinfo_tbl = NULL;

219     /* Bind fanout */
220     for (i = 0; i < dccps->dccps_bind_fanout_size; i++) {
221         ASSERT(dccps->dccps_bind_fanout[i].df_dccp == NULL);
222         mutex_destroy(&dccps->dccps_bind_fanout[i].df_lock);
223     }
224     kmem_free(dccps->dccps_bind_fanout, dccps->dccps_bind_fanout_size *
225     sizeof (dccp_df_t));
226     dccps->dccps_bind_fanout = NULL;

228     kmem_free(dccps, sizeof (*dccps));
229 }

231 /* /dev/dccp */
232 static int
233 dccp_openv4(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
234 {
235     cmn_err(CE_NOTE, "dccp.c: dccp_openv4\n");

237     return (ENOTSUP);
238 }

240 /* /dev/dccp6 */
241 static int
242 dccp_openv6(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
243 {
244     cmn_err(CE_NOTE, "dccp.c: dccp_openv6\n");

246     return (ENOTSUP);
247 }

249 /*
250 * IXA notify
251 */
252 static void
253 dccp_notify(void *arg, ip_xmit_attr_t *ixa, ix_notify_type_t ntype,
254     ix_notify_arg_t narg)
255 {
256     cmn_err(CE_NOTE, "dccp.c: dccp_notify");
257 }

259 /*

```



```

260 * Build the template headers.
261 */
262 int
263 dccp_build_hdrs(dccp_t *dccp)
264 {
265     dccp_stack_t    *dccps = dccp->dccp_dccps;
266     conn_t          *connp = dccp->dccp_connp;
267     dccpha_t        *dccpha;
268     uint32_t         cksum;
269     char             buf[DCCP_MAX_HDR_LENGTH];
270     uint_t           buflen;
271     uint_t           ulplen = 12;
272     uint_t           extralen = 0;
273     int              error;
274
275     cmn_err(CE_NOTE, "dccp.c: dccp_build_hdrs");
276
277     buflen = connp->conn_ht_ulp_len;
278     if (buflen != 0) {
279         cmn_err(CE_NOTE, "buflen != 0");
280         bcopy(connp->conn_ht_ulp, buf, buflen);
281         extralen -= buflen - ulplen;
282         ulplen = buflen;
283     }
284
285     mutex_enter(&connp->conn_lock);
286     error = conn_build_hdr_template(connp, ulplen, extralen,
287         &connp->conn_laddr_v6, &connp->conn_faddr_v6, connp->conn_flowinfo);
288     mutex_exit(&connp->conn_lock);
289     if (error != 0) {
290         cmn_err(CE_NOTE, "conn_build_hdr_template failed");
291         return (error);
292     }
293
294     dccpha = (dccpha_t *)connp->conn_ht_ulp;
295     dccp->dccp_dccpha = dccpha;
296
297     if (buflen != 0) {
298         bcopy(buf, connp->conn_ht_ulp, buflen);
299     } else {
300         dccpha->dha_sum = 0;
301         dccpha->dha_lport = connp->conn_lport;
302         dccpha->dha_fport = connp->conn_fport;
303     }
304
305     cksum = sizeof (dccpha_t) + connp->conn_sum;
306     cksum = (cksum >> 16) + (cksum & 0xFFFF);
307     dccpha->dha_sum = htons(cksum);
308     dccpha->dha_offset = 7;
309     dccpha->dha_x = 1;
310
311     return (0);
312 }
313
314 /*
315 * DCCP write service routine.
316 */
317 static void
318 dccp_wsrv(queue_t *q)
319 {
320     /* XXX:DCCP */
321 }
322
323 /*
324 * Common create function for streams and sockets.
325 */

```

```

326 conn_t *
327 dccp_create_common(cred_t *credp, boolean_t isv6, boolean_t issocket,
328     int *errorp)
329 {
330     conn_t          *connp;
331     dccp_t          *dccp;
332     dccp_stack_t    *dccps;
333     netstack_t      *ns;
334     squeue_t        *sqp;
335     zoneid_t        zoneid;
336
337     cmn_err(CE_NOTE, "dccp.c: dccp_create_common\n");
338
339     ASSERT(errorp != NULL);
340
341     *errorp = secpolicy_basic_net_access(credp);
342     if (*errorp != 0) {
343         return (NULL);
344     }
345
346     /*
347      * Find the right netstack
348      */
349     ns = netstack_find_by_cred(credp);
350     ASSERT(ns != NULL);
351     dccps = ns->netstack_dccp;
352     ASSERT(dccps != NULL);
353
354     /*
355      * XXX
356      */
357     if (ns->netstack_stackid != GLOBAL_NETSTACKID) {
358         zoneid = GLOBAL_ZONEID;
359     } else {
360         zoneid = crgetzoneid(credp);
361     }
362
363     sqp = IP_SQUEUE_GET((uint_t)gethrtime());
364     connp = (conn_t *)dccp_get_conn(sqp, dccps);
365     netstack_rele(dccps->dccps_netstack);
366     if (connp == NULL) {
367         *errorp = ENOSR;
368         return (NULL);
369     }
370     ASSERT(connp->conn_ixa->ixa_protocol == connp->conn_proto);
371
372     connp->conn_sqp = sqp;
373     connp->conn_initial_sqp = connp->conn_sqp;
374     connp->conn_ixa->ixa_sqp = connp->conn_sqp;
375     dccp = connp->conn_dccp;
376
377     /* Setting flags for ip output */
378     connp->conn_ixa->ixa_flags |= IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
379         IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO;
380
381     ASSERT(connp->conn_proto == IPPROTO_DCCP);
382     ASSERT(connp->conn_dccp == dccp);
383     ASSERT(dccp->dccp_connp == connp);
384
385     if (isv6) {
386         connp->conn_ixa->ixa_src_preferences = IPV6_PREFER_SRC_DEFAULT;
387         connp->conn_ipversion = IPV6_VERSION;
388         connp->conn_family = AF_INET6;
389         /* XXX mms, ttl */
390     } else {
391         connp->conn_ipversion = IPV4_VERSION;

```

```

392     connp->conn_family = AF_INET;
393     /* XXX mms, ttl */
394 }
395 connp->conn_xmit_ipp.ipp_unicast_hops = connp->conn_default_ttl;

397 crhold(credp);
398 connp->conn_cred = credp;
399 connp->conn_cpid = curproc->p_pid;
400 connp->conn_open_time = ddi_get_lbolt64();

402 ASSERT(!(connp->conn_ixa->ixa_free_flags & IXA_FREE_CRED));
403 connp->conn_ixa->ixa_cred = credp;
404 connp->conn_ixa->ixa_cpid = connp->conn_cpid;

406 connp->conn_zoneid = zoneid;
407 connp->conn_zone_is_global = (crgetzoneid(credp) == GLOBAL_ZONEID);
408 connp->conn_ixa->ixa_zoneid = zoneid;
409 connp->conn_mlp_type = mlptSingle;

412 dccp->dccp_dccps = dccps;
413 dccp->dccp_state = DCCPS_CLOSED;

415 ASSERT(connp->conn_netstack == dccps->dccps_netstack);
416 ASSERT(dccp->dccp_dccps == dccps);

418 /* XXX rcvbuf, sndbuf etc */

420 SOCK_CONNID_INIT(dccp->dccp_connid);
421 dccp_init_values(dccp, NULL);

423     return (connp);
424 }

426 /*
427  * Common close function for streams and sockets.
428  */
429 void
430 dccp_close_common(conn_t *connp)
431 {
432     dccp_t      *dccp = connp->conn_dccp;
433     boolean_t   conn_ioctl_cleanup_reqd = B_FALSE;

435     ASSERT(connp->conn_ref >= 2);

437     mutex_enter(&connp->conn_lock);
438     connp->conn_state_flags |= CONN_CLOSING;
439     if (connp->conn_oper_pending_ill != NULL) {
440         conn_ioctl_cleanup_reqd = B_TRUE;
441     }

443     CONN_INC_REF_LOCKED(connp);
444     mutex_exit(&connp->conn_lock);

446     //ipcl_conn_destroy(connp);
447 }

449 /*
450  * Common bind function.
451  */
452 int
453 dccp_do_bind(conn_t *connp, struct sockaddr *sa, socklen_t len, cred_t *cr,
454             boolean_t bind_to_req_port_only)
455 {
456     dccp_t      *dccp = connp->conn_dccp;
457     int          error;

```

```

459     cmn_err(CE_NOTE, "dccp.c: dccp_do_bind");

461     if (dccp->dccp_state >= DCCPS_BOUND) {
462         if (connp->conn_debug) {
463             (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
464                 "dccp_bind: bad state, %d", dccp->dccp_state);
465         }
466         return (-TOUTSTATE);
467     }

469     error = dccp_bind_check(connp, sa, len, cr, bind_to_req_port_only);
470     if (error != 0) {
471         return (error);
472     }

474     ASSERT(dccp->dccp_state == DCCPS_LISTEN);
475     /* XXX dccp_conn_req_max = 0 */

477     return (0);
478 }

480 /*
481  * Common unbind function.
482  */
483 int
484 dccp_do_unbind(conn_t *connp)
485 {
486     dccp_t      *dccp = connp->conn_dccp;

488     cmn_err(CE_NOTE, "dccp.c: dccp_do_unbind");

490     switch (dccp->dccp_state) {
491     case DCCPS_BOUND:
492         break;
493     case DCCPS_LISTEN:
494         break;
495     default:
496         return (-TOUTSTATE);
497     }

498     /* XXX:DCCP */

500     return (0);
501 }

503 /*
504  * Common listen function.
505  */
506 int
507 dccp_do_listen(conn_t *connp, struct sockaddr *sa, socklen_t len,
508               int backlog, cred_t *cr, boolean_t bind_to_req_port_only)
509 {
510     dccp_t      *dccp = connp->conn_dccp;
511     dccp_stack_t *dccps = dccp->dccp_dccps;
512     int32_t      oldstate;
513     int          error;

515     cmn_err(CE_NOTE, "dccp.c: dccp_do_listen");

517     /* All Solaris components should pass a cred for this operation */
518     ASSERT(cr != NULL);

520     if (dccp->dccp_state >= DCCPS_BOUND) {

522         if ((dccp->dccp_state == DCCPS_BOUND ||
523             dccp->dccp_state == DCCPS_LISTEN) && backlog > 0) {

```

```

524         goto do_listen;
525     }
526     cmn_err(CE_NOTE, "DCCPS_BOUND, bad state");

528     if (connp->conn_debug) {
529         (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
530             "dccp_listen: bad state, %d", dccp->dccp_state);
531     }
532     return (-TOUTSTATE);
533 } else {
534     if (sa == NULL) {
535         sin6_t  addr;
536         sin6_t  *sin6;
537         sin_t   *sin;

539         ASSERT(IPCL_IS_NONSTR(connp));

541         if (connp->conn_family == AF_INET) {
542             len = sizeof (sin_t);
543             sin = (sin_t *)&addr;
544             *sin = sin_null;
545             sin->sin_family = AF_INET;
546         } else {
547             ASSERT(connp->conn_family == AF_INET6);

549             len = sizeof (sin6_t);
550             sin6 = (sin6_t *)&addr;
551             *sin6 = sin6_null;
552             sin6->sin6_family = AF_INET6;
553         }

555         sa = (struct sockaddr *)&addr;
556     }

558     error = dccp_bind_check(connp, sa, len, cr,
559         bind_to_req_port_only);
560     if (error != 0) {
561         cmn_err(CE_NOTE, "dccp_bind_check failed");
562         return (error);
563     }
564     /* Fall through and do the fanout insertion */
565 }

567 do_listen:
568     ASSERT(dccp->dccp_state == DCCPS_BOUND ||
569         dccp->dccp_state == DCCPS_LISTEN);

571     /* XXX backlog */

573     connp->conn_rcv = dccp_input_listener_unbound;

575     /* Insert into the classifier table */
576     error = ip_laddr_fanout_insert(connp);
577     if (error != 0) {
578         /* Error - undo the bind */
579         oldstate = dccp->dccp_state;
580         dccp->dccp_state = DCCPS_CLOSED;

582         connp->conn_bound_addr_v6 = ipv6_all_zeros;

584         connp->conn_laddr_v6 = ipv6_all_zeros;
585         connp->conn_saddr_v6 = ipv6_all_zeros;
586         connp->conn_ports = 0;

588         if (connp->conn_anon_port) {
589             zone_t  *zone;

```

```

591         zone = crgetzone(cr);
592         connp->conn_anon_port = B_FALSE;
593         (void) tsol_mlp_anon(zone, connp->conn_mlp_type,
594             connp->conn_proto, connp->conn_lport, B_FALSE);
595     }
596     connp->conn_mlp_type = mlptSingle;

598     /* XXX dccp_bind_hash_remove */

600     return (error);
601 } else {
602     /* XXX connection limits */
603 }

605     return (error);
606 }

608 /*
609  * Common connect function.
610  */
611 int
612 dccp_do_connect(conn_t *connp, const struct sockaddr *sa, socklen_t len,
613     cred_t *cr, pid_t pid)
614 {
615     dccp_t          *dccp = connp->conn_dccp;
616     dccp_stack_t    *dccps = dccp->dccp_dccps;
617     ip_xmit_attr_t  *ixa = connp->conn_ixa;
618     sin_t           *sin = (sin_t *)sa;
619     sin6_t          *sin6 = (sin6_t *)sa;
620     ipaddr_t        *dstaddrp;
621     in_port_t       dstport;
622     int32_t         oldstate;
623     uint_t          srcid;
624     int             error;

626     cmn_err(CE_NOTE, "dccp.c: dccp_do_connect");

628     oldstate = dccp->dccp_state;

630     switch (len) {
631     case sizeof (sin_t):
632         sin = (sin_t *)sa;
633         if (sin->sin_port == 0) {
634             return (-TBADADDR);
635         }
636         if (connp->conn_ipv6_v6only) {
637             return (EAFNOSUPPORT);
638         }
639         break;

641     case sizeof (sin6_t):
642         sin6 = (sin6_t *)sa;
643         if (sin6->sin6_port == 0) {
644             return (-TBADADDR);
645         }
646         break;

648     default:
649         return (EINVAL);
650     }

652     if (connp->conn_family == AF_INET6 &&
653         connp->conn_ipversion == IPV6_VERSION &&
654         IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
655         if (connp->conn_ipv6_v6only) {

```

```

656         return (EADDRNOTAVAIL);
657     }
659     connp->conn_ipversion = IPV4_VERSION;
660 }
662 switch (dccp->dccp_state) {
663 case DCCPS_LISTEN:
664     if (IPCL_IS_NONSTR(connp)) {
665         return (EOPNOTSUPP);
666     }
668 case DCCPS_CLOSED:
669     /* XXX */
670     break;
672 default:
673     return (-TOUTSTATE);
674 }
676 if (connp->conn_cred != cr) {
677     crhold(cr);
678     crfree(connp->conn_cred);
679     connp->conn_cred = cr;
680 }
681 connp->conn_cpid = pid;
683 ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
684 ixa->ixa_cred = cr;
685 ixa->ixa_cpid = pid;
687 if (is_system_labeled()) {
688     ip_xmit_attr_restore_tsl(ixa, ixa->ixa_cred);
689 }
691 if (connp->conn_family == AF_INET6) {
692     if (!IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
693         error = dccp_connect_ipv6(dccp, &sin6->sin6_addr,
694             sin6->sin6_port, sin6->sin6_flowinfo,
695             sin6->__sin6_src_id, sin6->sin6_scope_id);
696     } else {
697         /* XXX */
698     }
699 } else {
700     dstaddrp = &sin->sin_addr.s_addr;
701     dstport = sin->sin_port;
702     srcid = 0;
703     error = dccp_connect_ipv4(dccp, dstaddrp, dstport, srcid);
704 }
706 if (error != 0) {
707     goto connect_failed;
708 }
710 /* XXX cluster */
712 //DCCPS_BUMP_MIB(dccps, dccpActiveOpens);
714 return (0);
716 connect_failed:
717     cmn_err(CE_NOTE, "dccp_do_connect failed");
719     connp->conn_faddr_v6 = ipv6_all_zeros;
720     connp->conn_fport = 0;
721     dccp->dccp_state = oldstate;

```

```

723     return (error);
724 }
726 /*
727  * Init values of a connection.
728  */
729 void
730 dccp_init_values(dccp_t *dccp, dccp_t *parent)
731 {
732     conn_t         *connp = dccp->dccp_connp;
733     dccp_stack_t   *dccps = dccp->dccp_dccps;
735     connp->conn_mlp_type = mlptSingle;
736 }
738 void *
739 dccp_get_conn(void *arg, dccp_stack_t *dccps)
740 {
741     dccp_t         *dccp = NULL;
742     conn_t         *connp;
743     squeue_t       *sqp = (squeue_t *)arg;
744     netstack_t     *ns;
746     /* XXX timewait */
748     connp = ipcl_conn_create(IPCL_DCCPCONN, KM_NOSLEEP,
749         dccps->dccps_netstack);
750     if (connp == NULL) {
751         return (NULL);
752     }
754     dccp = connp->conn_dccp;
755     dccp->dccp_dccps = dccps;
757     connp->conn_recv = dccp_input_data;
758     connp->conn_recvicmp = dccp_icmp_input;
759     connp->conn_verifyicmp = dccp_verifyicmp;
761     connp->conn_ixa->ixa_notify = dccp_notify;
762     connp->conn_ixa->ixa_notify_cookie = dccp;
764     return ((void *)connp);
765 }
767 /*
768  * IPv4 connect.
769  */
770 static int
771 dccp_connect_ipv4(dccp_t *dccp, ipaddr_t *dstaddrp, in_port_t dstport,
772     uint_t srcid)
773 {
774     conn_t         *connp = dccp->dccp_connp;
775     dccp_stack_t   *dccps = dccp->dccp_dccps;
776     ipaddr_t       dstaddr = *dstaddrp;
777     uint16_t       lport;
778     int            error;
780     cmn_err(CE_NOTE, "dccp.c: dccp_connect_ipv4");
782     ASSERT(connp->conn_ipversion == IPV4_VERSION);
784     if (dstaddr == INADDR_ANY) {
785         dstaddr = htonl(INADDR_LOOPBACK);
786         *dstaddrp = dstaddr;
787     }

```

```

789     if (srcid != 0 && connp->conn_laddr_v4 == INADDR_ANY) {
790         ip_srcid_find_id(srcid, &connp->conn_laddr_v6,
791             IPCL_ZONEID(connp), dccps->dccps_netstack);
792         connp->conn_saddr_v6 = connp->conn_laddr_v6;
793     }

795     IN6_IPADDR_TO_V4MAPPED(dstaddr, &connp->conn_faddr_v6);
796     connp->conn_fport = dstport;

798     if (dccp->dccp_state == DCCPS_CLOSED) {
799         lport = dccp_update_next_port(dccps->dccps_next_port_to_try,
800             dccp, B_TRUE);
801         lport = dccp_bindi(dccp, lport, &connp->conn_laddr_v6, 0,
802             B_TRUE, B_FALSE, B_FALSE);

804         if (lport == 0) {
805             return (-TNOADDR);
806         }
807     }

809     error = dccp_set_destination(dccp);
810     if (error != 0) {
811         return (error);
812     }

814     /*
815      * Don't connect to oneself.
816      */
817     if (connp->conn_faddr_v4 == connp->conn_laddr_v4 &&
818         connp->conn_fport == connp->conn_lport) {
819         return (-TBADADDR);
820     }

822     /* XXX state */

824     return (ipcl_conn_insert_v4(connp));
825 }

827 /*
828  * IPv6 connect.
829  */
830 static int
831 dccp_connect_ipv6(dccp_t *dccp, in6_addr_t *dstaddrp, in_port_t dstport,
832     uint32_t flowinfo, uint_t srcid, uint32_t scope_id)
833 {
834     cmn_err(CE_NOTE, "dccp.c: dccp_connect_ipv6");

836     return (0);
837 }

839 /*
840  * Set the ports via conn_connect and build the template
841  * headers.
842  */
843 int
844 dccp_set_destination(dccp_t *dccp)
845 {
846     conn_t         *connp = dccp->dccp_connp;
847     dccp_stack_t   *dccps = dccp->dccp_dccps;
848     iulp_t         uinfo;
849     uint32_t       flags;
850     int            error;

852     flags = IPDF_LSO | IPDF_ZCOPY;
853     flags |= IPDF_UNIQUE_DCE;

```

```

855     mutex_enter(&connp->conn_lock);
856     error = conn_connect(connp, &uinfo, flags);
857     mutex_exit(&connp->conn_lock);
858     if (error != 0) {
859         cmn_err(CE_NOTE, "conn_connect failed");
860         return (error);
861     }

863     error = dccp_build_hdrs(dccp);
864     if (error != 0) {
865         cmn_err(CE_NOTE, "dccp_build_hdrs failed");
866         return (error);
867     }

869     /* XXX */

871     mutex_enter(&connp->conn_lock);
872     connp->conn_state_flags &= ~CONN_INCIPIENT;
873     mutex_exit(&connp->conn_lock);

875     return (0);
876 }
877 #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp/dccp.conf

1

913 Wed Jun 13 12:04:29 2012

new/usr/src/uts/common/inet/dccp/dccp.conf

%B

```
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 #
25
26 name="dccp" parent="pseudo" instance=0;
27 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp6.conf

1

914 Wed Jun 13 12:04:30 2012

new/usr/src/uts/common/inet/dccp/dccp6.conf

%B

```
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 #
25
26 name="dccp6" parent="pseudo" instance=0;
27 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp6ddi.c

1

1578 Wed Jun 13 12:04:31 2012

new/usr/src/uts/common/inet/dccp/dccp6ddi.c

%B

```
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 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #include <sys/types.h>
27 #include <sys/conf.h>
28 #include <sys/modctl.h>
29 #include <inet/common.h>
30 #include <inet/ip.h>

32 #define INET_NAME "dccp6"
33 #define INET_DEVSTRTAB dccpinfov6
34 #define INET_DEVDESC "DCCP6 STREAMS driver"
35 #define INET_DEVMINOR 0
36 #define INET_DEVMTFLAGS (D_MP|_D_DIRECT)

38 #include "../inetddi.c"

40 int
41 _init(void)
42 {
43     /*
44      * device initialization happens when the actual code containing
45      * module (/kernel/drv/ip) is loaded, and driven from ip_ddi_init()
46      */
47     return (mod_install(&modlinkage));
48 }

50 int
51 _fini(void)
52 {
53     return (mod_remove(&modlinkage));
54 }

56 int
57 _info(struct modinfo *modinfop)
58 {
59     return (mod_info(&modlinkage, modinfop));
60 }
61 #endif /* ! codereview */
```


new/usr/src/uts/common/inet/dccp/dccp_bind.c

1

```
*****
10340 Wed Jun 13 12:04:32 2012
new/usr/src/uts/common/inet/dccp/dccp_bind.c
%B
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */

27 /*
28  * Copyright 2012 David Hoepfner. All rights reserved.
29  */

31 /*
32  * This file contains function related to binding.
33  */

35 #include <sys/types.h>
36 #include <sys/stream.h>
37 #include <sys/strsun.h>
38 #include <sys/strsubr.h>
39 #include <sys/stropts.h>
40 #include <sys/strlog.h>
41 #define _SUN_TPI_VERSION 2
42 #include <sys/tihdr.h>
43 #include <sys/suntpi.h>
44 #include <sys/xti_inet.h>
45 #include <sys/queue_impl.h>
46 #include <sys/queue.h>
47 #include <sys/tsol/tnet.h>

49 #include <inet/common.h>
50 #include <inet/ip.h>
51 #include <inet/proto_set.h>

53 #include <sys/cmn_err.h>

55 #include "dccp_impl.h"

57 /* Setable in /etc/system */
58 static uint32_t dccp_random_anon_port = 1;

60 static int dccp_bind_select_lport(dccp_t *, in_port_t *, boolean_t,
61     cred_t *);
```

new/usr/src/uts/common/inet/dccp/dccp_bind.c

2

```
63 void
64 dccp_bind_hash_insert(dccp_df_t *tbf, dccp_t *dccp, int caller_holds_lock)
65 {
66     conn_t *connp = dccp->dccp_connp;
67     conn_t *connxt;
68     dccp_t **dccpp;
69     dccp_t *dccpnext;
70     dccp_t *dccphash;

72     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_hash_insert");

74     /* XXX:DCCP */

76     dccpp = &tbf->df_dccp;
77     if (!caller_holds_lock) {
78         mutex_enter(&tbf->df_lock);
79     } else {
80         ASSERT(MUTEX_HELD(&tbf->df_lock));
81     }
82     dccphash = dccpp[0];
83     dccpnext = NULL;

85     if (dccphash != NULL) {
86         /* XXX:DCCP */
87     }

89 insert:
90     dccp->dccp_bind_hash_port = dccpnext;
91     dccp->dccp_bind_hash = dccphash;
92     dccp->dccp_ptpbhn = dccpp;
93     dccpp[0] = dccp;

95     if (!caller_holds_lock) {
96         mutex_exit(&tbf->df_lock);
97     }
98 }

100 void
101 dccp_bind_hash_remove(dccp_t *dccp)
102 {
103 }

105 /*
106  * Check for a valid address and get a local port.
107  */
108 int
109 dccp_bind_check(conn_t *connp, struct sockaddr *sa, socklen_t len, cred_t *cr,
110     boolean_t bind_to_req_port_only)
111 {
112     dccp_t *dccp = connp->conn_dccp;
113     ip_stack_t *ips = connp->conn_netstack->netstack_ip;
114     ip_xmit_attr_t *ixa = connp->conn_ixa;
115     sin_t *sin;
116     sin6_t *sin6;
117     ipaddr_t v4addr;
118     in6_addr_t v6addr;
119     ip_laddr_t laddr_type = IPVL_UNICAST_UP;
120     zoneid_t zoneid = IPCL_ZONEID(connp);
121     in_port_t requested_port;
122     uint_t scopeid = 0;
123     int error;

125     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_check");

127     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  }

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  }

156  error = proto_verify_ip_addr(connp->conn_family, sa, len);
157  if (error != 0) {
158      return (error);
159  }

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;

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          }

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;

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  }

206  if (laddr_type == IPV6_BAD) {
207      return (EADDRNOTAVAIL);
208  }

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  }

220  connp->conn_laddr_v6 = v6addr;
221  connp->conn_saddr_v6 = v6addr;

223  bind_to_req_port_only = requested_port != 0 && bind_to_req_port_only;

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  }

233  return (error);
234 }

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;

251      cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_select_lport");

253      ASSERT(cr != NULL);

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 */
```

new/usr/src/uts/common/inet/dccp/dccp_impl.h

1

```
*****
7135 Wed Jun 13 12:04:32 2012
new/usr/src/uts/common/inet/dccp/dccp_impl.h
%B
*****
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */
12 /*
13  * Copyright 2012 David Hoepfner. All rights reserved.
14 */
16 #ifndef _INET_DCCP_IMPL_H
17 #define _INET_DCCP_IMPL_H
19 #include <sys/int_types.h>
20 #include <sys/netstack.h>
21 #include <sys/socket.h>
22 #include <sys/socket_proto.h>
24 #include <netinet/in.h>
25 #include <netinet/ip6.h>
26 #include <netinet/dccp.h>
28 #include <inet/common.h>
29 #include <inet/ip.h>
30 #include <inet/ip6.h>
31 #include <inet/optcom.h>
32 #include <inet/tunables.h>
34 #include "dccp_stack.h"
36 #ifdef __cplusplus
37 extern "C" {
38 #endif
40 #ifdef _KERNEL
42 #define DCCP_MOD_ID 5999 /* XXX */
44 #define DCCP_XMIT_LOWATER (4 * 1024)
45 #define DCCP_XMIT_HIWATER 49152
46 #define DCCP_RECV_LOWATER (2 * 1024)
47 #define DCCP_RECV_HIWATER 128000
49 /*
50  * Bind hash array size and hash function.
51  */
52 #define DCCP_BIND_FANOUT_SIZE 128
53 #define DCCP_BIND_HASH(lport, size) ((ntohs((uint16_t)lport)) & (size - 1))
56 #define DCCP_HDR_LENGTH(dccph) (dccph_t *)dccph->dh_offset
57 #define DCCP_MAX_HDR_LENGTH 1020
58 #define DCCP_MIN_HEADER_LENGTH 12
60 /* Packet types (RFC 4340, Section 5.1.) */
61 #define DCCP_PKT_REQUEST 0
```

new/usr/src/uts/common/inet/dccp/dccp_impl.h

2

```
62 #define DCCP_PKT_RESPONSE 1
63 #define DCCP_PKT_DATA 2
64 #define DCCP_PKT_ACK 3
65 #define DCCP_PKT_DATAACK 4
66 #define DCCP_PKT_CLOSEREQ 5
67 #define DCCP_PKT_CLOSE 6
68 #define DCCP_PKT_RESET 7
69 #define DCCP_PKT_SYNC 8
70 #define DCCP_PKT_SYNCAK 9
72 /* Generic protocol header (RFC 4340, Section 5.1.) */
73 typedef struct dccphdr_s {
74     uint8_t dh_lport[2];
75     uint8_t dh_fport[2];
76     uint8_t dh_offset;
77     uint8_t dh_ccval:4;
78     uint8_t dh_cscov:4;
79     uint8_t db_sum[2];
80     uint8_t dh_reserved:3;
81     uint8_t dh_type:4;
82     uint8_t dh_x:1;
83     uint8_t dh_res_seq;
84     uint8_t dh_seq[2];
85 } dccph_t;
88 /* Generic protocol header aligned (RFC 4340, Section 5.1.) */
89 typedef struct dccphdra_s {
90     in_port_t dha_lport; /* Source port */
91     in_port_t dha_fport; /* Destination port */
92     uint8_t dha_offset; /* Data offset */
93     uint8_t dha_ccval:4; /* */
94     uint8_t dha_cscov:4; /* */
95     uint16_t dha_sum; /* Checksum */
96     uint8_t dha_x:1; /* Reserved */
97     uint8_t dha_type:4; /* Packet type */
98     uint8_t dha_reserved:3; /* Header type */
99     uint8_t dha_res_seq;
100     uint16_t dha_seq; /* Partial sequence number */
101 } dccpha_t;
103 typedef struct dccphdra_ext_s {
104     uint32_t dha_ext_seq;
105 } dccpha_ext_t;
107 /* Acknowledgement number */
108 typedef struct dccphdra_ack {
109     uint16_t dha_ack_reserved;
110     uint16_t dha_ack_high;
111     uint32_t dha_ack_low;
112 } dccpha_ack_t;
114 typedef struct dccphdra_srv {
115     uint32_t dha_srv_code;
116 } dccpha_srv_t;
118 typedef struct dccphdra_reset {
119     uint8_t dha_reset_code;
120     uint8_t dha_reset_data[3];
121 } dccpha_reset_t;
123 /* Internal DCCP structure */
124 typedef struct dccp_s {
126     conn_t *dccp_connp; /* Backpointer to conn_t */
127     dccp_stack_t *dccp_dccps; /* Backpointer to dccp_stack_t */
```

```

129     uint32_t         dccp_state;

131     /* Bind related */
132     struct dccp_s     *dccp_bind_hash;      /* Bind hash chain */
133     struct dccp_s     *dccp_bind_hash_port; /* Bound to the same port */
134     struct dccp_s     **dccp_ptpbhn;

136     struct dccphdra_s *dccp_dccpha;        /* Template header */

138     mblk_t           *dccp_xmit_head;

140     sock_connid_t    dccp_connid;
141 } dccp_t;

143 #define dccps_smallest_nonpriv_port    dccps_propinfo_tbl[0].prop_cur_uval
144 #define dccps_smallest_anon_port      dccps_propinfo_tbl[1].prop_cur_uval
145 #define dccps_largest_anon_port       dccps_propinfo_tbl[2].prop_cur_uval

147 #define dccps_dbg                       dccps_propinfo_tbl[4].prop_cur_uval
148 #define dccps_rst_sent_rate_enabled    dccps_propinfo_tbl[5].prop_cur_uval
149 #define dccps_rst_sent_rate            dccps_propinfo_tbl[6].prop_cur_uval

151 typedef struct dccp_df_s {
152     struct dccp_s     *df_dccp;
153     kmutex_t          df_lock;
154     uchar_t           df_pad[TF_CACHEL_PAD - (sizeof (dccp_t *) +
155                                     sizeof (kmutex_t))];
156 } dccp_df_t;

158 extern struct qinit dccp_rinitv4, dccp_rinitv6;

160 extern optdb_obj_t    dccp_opt_obj;
161 extern uint_t         dccp_max_optsize;

163 /*
164  * Functions in dccp.c
165  */
166 extern int            dccp_build_hdrs(dccp_t *);
167 extern conn_t         *dccp_create_common(cred_t *, boolean_t, boolean_t, int *);
168 extern void           dccp_close_common(conn_t *);
169 extern int            dccp_do_bind(conn_t *, struct sockaddr *, socklen_t, cred_t *,
170                                 boolean_t);
171 extern int            dccp_do_unbind(conn_t *);
172 extern int            dccp_do_listen(conn_t *, struct sockaddr *, socklen_t, int,
173                                 cred_t *, boolean_t);
174 extern int            dccp_do_connect(conn_t *, const struct sockaddr *, socklen_t,
175                                 cred_t *, pid_t);
176 extern void           dccp_init_values(dccp_t *, dccp_t *);
177 extern void           *dccp_get_conn(void *, dccp_stack_t *);
178 extern int            dccp_set_destination(dccp_t *dccp);

180 /*
181  * Bind related functions in dccp_bind.c
182  */
183 extern void           dccp_bind_hash_insert(dccp_df_t *, dccp_t *, int);
184 extern void           dccp_bind_hash_remove(dccp_t *);
185 extern int            dccp_bind_check(conn_t *, struct sockaddr *, socklen_t, cred_t *,
186                                 boolean_t);
187 extern in_port_t     dccp_bindi(dccp_t *, in_port_t, const in6_addr_t *, int,
188                                 boolean_t, boolean_t, boolean_t);
189 extern in_port_t     dccp_update_next_port(in_port_t, const dccp_t *, boolean_t);

191 /*
192  * MIB-II and kstat related functions.
193  */

```

```

194 extern mblk_t        *dccp_snmp_get(queue_t *, mblk_t *);

196 /*
197  * Socket related functions in dccp_socket.c
198  */
199 extern sock_lower_handle_t dccp_create(int, int, int, sock_downcalls_t **,
200                                     uint_t *, int *, int, cred_t *);
201 extern int              dccp_fallback(sock_lower_handle_t, queue_t *, boolean_t,
202                                     so_proto_quiesced_cb_t, sock_quiesce_arg_t *);

204 /*
205  * Input path related functions in dccp_input.c
206  */
207 extern void            dccp_icmp_input(void *, mblk_t *, void *, ip_rcv_attr_t *);
208 extern void            dccp_input_data(void *, mblk_t *, void *, ip_rcv_attr_t *);
209 extern void            dccp_rsrv(queue_t *);
210 extern void            dccp_input_listener_unbound(void *, mblk_t *, void *,
211                                                    ip_rcv_attr_t *);
212 extern boolean_t       dccp_verifyicmp(conn_t *, void *, icmp_h_t *, icmp6_t *,
213                                       ip_rcv_attr_t *);
214 /*
215  * Output path related functions in dccp_output.c
216  */
217 extern void            dccp_wput(queue_t *, mblk_t *);
218 extern void            dccp_xmit_listeners_reset(mblk_t *, ip_rcv_attr_t *,
219                                                  ip_stack_t *, conn_t *);
220 extern void            dccp_send_synack(void *, mblk_t *, void *, ip_rcv_attr_t *);
221 extern mblk_t         *dccp_xmit_mp(dccp_t *, mblk_t *, int32_t, int32_t *,
222                                     mblk_t **, uint32_t, boolean_t, uint32_t *, boolean_t);
223 extern mblk_t         *dccp_generate_packet(conn_t *, mblk_t *);
224 /*
225  * Options related functions in dccp_opt_data.c
226  */
227 extern int            dccp_opt_get(conn_t *, int, int, uchar_t *);
228 extern int            dccp_opt_set(conn_t *, uint_t, int, int, uint_t, uchar_t *,
229                                     uint_t *, uchar_t *, void *, cred_t *);

231 /*
232  * dccp_tpi.c
233  */
234 extern void           dccp_err_ack(dccp_t *, mblk_t *, int, int);
235 extern void           dccp_tpi_connect(dccp_t *, mblk_t *);
236 extern int            dccp_tpi_close(queue_t *, int);
237 extern int            dccp_tpi_opt_get(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
238 extern int            dccp_tpi_opt_set(queue_t *, uint_t, int, int, uint_t, uchar_t *,
239                                     uint_t *, uchar_t *, void *, cred_t *);

241 #endif /* _KERNEL */

243 #ifdef __cplusplus
244 }
245 #endif

247 #endif /* _INET_DCCP_IMPL_H */
248 #endif /* ! codereview */

```

```

*****
8662 Wed Jun 13 12:04:33 2012
new/usr/src/uts/common/inet/dccp/dccp_input.c
%B
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */

27 /*
28  * Copyright 2012 David Hoepfner. All rights reserved.
29  */

31 #include <sys/types.h>
32 #include <sys/stream.h>
33 #include <sys/strsun.h>
34 #include <sys/strsubr.h>
35 #include <sys/stropts.h>
36 #include <sys/strlog.h>
37 #define _SUN_TPI_VERSION 2
38 #include <sys/tihdr.h>
39 #include <sys/suntpi.h>
40 #include <sys/xti_inet.h>
41 #include <sys/squeue_impl.h>
42 #include <sys/squeue.h>
43 #include <sys/tsol/tnet.h>

45 #include <inet/common.h>
46 #include <inet/ip.h>

48 #include <sys/cmn_err.h>

50 #include "dccp_impl.h"

52 static mblk_t *dccp_conn_create_v4(conn_t *, conn_t *, mblk_t *,
53 ip_recv_attr_t *);
54 static mblk_t *dccp_conn_create_v6(conn_t *, conn_t *, mblk_t *,
55 ip_recv_attr_t *);
56 static void dccp_input_listener(void *, mblk_t *, void *, ip_recv_attr_t *);

58 void
59 dccp_input_data(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
60 {
61     cmn_err(CE_NOTE, "dccp_input.c: dccp_input_data");

```

```

62 }

64 void
65 dccp_icmp_input(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
66 {
67     cmn_err(CE_NOTE, "dccp_input.c: dccp_icmp_input");
68 }

70 void
71 dccp_rsrv(queue_t *q)
72 {
73     cmn_err(CE_NOTE, "dccp_input.c: dccp_rsrv");
74 }

76 static mblk_t *
77 dccp_conn_create_v6(conn_t *lconnp, conn_t *connp, mblk_t *mp,
78 ip_recv_attr_t *ira)
79 {
80     return (NULL);
81 }

83 static mblk_t *
84 dccp_conn_create_v4(conn_t *lconnp, conn_t *connp, mblk_t *mp,
85 ip_recv_attr_t *ira)
86 {
87     dccp_t *ldccp = lconnp->conn_dccp;
88     dccp_t *dccp = connp->conn_dccp;
89     dccp_stack_t *dccps = dccp->dccp_dccps;
90     ipha_t *ipha;
91     mblk_t *tpi_mp;
92     sin_t *sin;

94     ASSERT(ira->ira_flags & IRAF_IS_IPV4);
95     ipha = (ipha_t *)mp->b_rptr;

97     connp->conn_ipversion = IPV4_VERSION;
98     IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &connp->conn_laddr_v6);
99     IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &connp->conn_faddr_v6);
100    connp->conn_saddr_v6 = connp->conn_laddr_v6;

102    sin = sin_null;
103    sin.sin_addr.s_addr = connp->conn_faddr_v4;
104    sin.sin_port = connp->conn_fport;
105    sin.sin_family = AF_INET;

107    if (lconnp->conn_recv_ancillary.crb_recvdstaddr) {
108        cmn_err(CE_NOTE, "ancillary");

110        sin_t *sind;

112        sind = sin_null;
113        sind.sin_addr.s_addr = connp->conn_laddr_v4;
114        sind.sin_port = connp->conn_lport;
115        sind.sin_family = AF_INET;

117        tpi_mp = mi_tpi_extconn_ind(NULL,
118 (char *)&sind, sizeof (sin_t), (char *)&dccp,
119 (t_scalar_t)sizeof (intptr_t), (char *)&sind,
120 sizeof (sin_t), (t_scalar_t) 1); /* XXX */

122    } else {
123        tpi_mp = mi_tpi_conn_ind(NULL,
124 (char *)&sin, sizeof (sin_t),
125 (char *)&dccp, (t_scalar_t)sizeof (intptr_t),
126 (t_scalar_t) 1); /* XXX */
127    }

```

```

129     return (tpi_mp);
130 }

132 static void
133 dccp_input_listener(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *ira)
134 {
135     conn_t      *lconnp = (conn_t *)arg;
136     econnp;
137     dccp_t      *listener = lconnp->conn_dccp;
138     *eager;
139     dccp_stack_t *dccps = listener->dccp_dccps;
140     ip_stack_t  *ipst = dccps->dccps_netstack->netstack_ip;
141     dccpha_t    *dccpha;
142     squeue_t    *new_sqp;
143     mblk_t      *tpi_mp;
144     mblk_t      *mpl;
145     uint_t      ip_hdr_len;
146     uint_t      type;
147     int         error;

149     cmn_err(CE_NOTE, "dccp_input.c: dccp_input_listener");

151     ip_hdr_len = ira->ira_ip_hdr_length;
152     dccpha = (dccpha_t *)&mp->b_rptr[ip_hdr_len];
153     type = (uint_t)dccpha->dha_type;

155     if (type != DCCP_PKT_REQUEST) {
156         cmn_err(CE_NOTE, "not request pkt");

158         /* XXX do something with a reset packet sent? */
159         freemsg(mp);
160         return;
161     }

163     /* XXX memory pressure */

165     /* XXX request defense */

167     /* XXX number of connections per listener */

169     ASSERT(ira->ira_sqp != NULL);
170     new_sqp = ira->ira_sqp;

172     econnp = (conn_t *)dccp_get_conn(arg2, dccps);
173     if (econnp == NULL) {
174         cmn_err(CE_NOTE, "econnp not found (eager)");
175         goto error2;
176     }

178     ASSERT(econnp->conn_netstack == lconnp->conn_netstack);
179     econnp->conn_sqp = new_sqp;
180     econnp->conn_initial_sqp = new_sqp;
181     econnp->conn_ixa->ixa_sqp = new_sqp;

183     econnp->conn_fport = dccpha->dha_lport;
184     econnp->conn_lport = dccpha->dha_fport;

186     error = conn_inherit_parent(lconnp, econnp);
187     if (error != 0) {
188         cmn_err(CE_NOTE, "conn_inherit_parent failed");
189         goto error3;
190     }

192     econnp->conn_ixa->ixa_src_generation = ipst->ips_src_generation;

```

```

195     ASSERT(OK_32PTR(mp->b_rptr));
196     ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION ||
197           IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);

199     if (lconnp->conn_family == AF_INET) {
200         ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION);
201         tpi_mp = dccp_conn_create_v4(lconnp, econnp, mp, ira);
202     } else {
203         tpi_mp = dccp_conn_create_v6(lconnp, econnp, mp, ira);
204     }

206     if (tpi_mp == NULL) {
207         cmn_err(CE_NOTE, "tpi_mo == NULL");
208         goto error3;
209     }

211     eager = econnp->conn_dccp;
212     SOCK_CONNID_INIT(eager->dccp_connid);

214     dccp_init_values(eager, listener);

216     ASSERT((econnp->conn_ixa->ixa_flags &
217           (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
218           IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO)) ==
219           (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
220           IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO));

222     if (ira->ira_cred != NULL) {
223         mblk_setcred(tpi_mp, ira->ira_cred, ira->ira_cpid);
224     }

226     if (IPCL_IS_NONSTR(lconnp)) {
227         econnp->conn_flags |= IPCL_NONSTR;
228     }

230     /* XXX dccps is right? */
231     dccp_bind_hash_insert(&dccps->dccps_bind_fanout[
232         DCCP_BIND_HASH(econnp->conn_lport, dccps->dccps_bind_fanout_size)],

234     SOCK_CONNID_BUMP(eager->dccp_connid);

236     error = dccp_set_destination(eager);
237     if (error != 0) {
238         cmn_err(CE_NOTE, "dccp_set_destination failed.");
239         dccp_bind_hash_remove(eager);
240         goto error3;
241     }

243     CONN_INC_REF(lconnp);

245 /*
246     mpl = dccp_xmit_mp(eager, eager->dccp_xmit_head, 0,
247         NULL, NULL, 0, B_FALSE, NULL, B_FALSE);
248 */
249     mpl = dccp_generate_packet(lconnp, mp);
250     if (mpl == NULL) {
251         cmn_err(CE_NOTE, "dccp_xmit_mp failed");

253         CONN_INC_REF(econnp);
254         goto error;
255     }

257     CONN_INC_REF(econnp);

259     error = ipcl_conn_insert(econnp);

```



```

260     if (error != 0) {
261         cmn_err(CE_NOTE, "ipcl_conn_insert(econnp) failed");
262         goto error;
263     }
265     freemsg(mp);
267     if (econnp->conn_sqp == lconnp->conn_sqp) {
268         (void) conn_ip_output(mpl, econnp->conn_ixa);
269         CONN_DEC_REF(econnp);
270     } else {
271         SQUEUE_ENTER_ONE(econnp->conn_sqp, mpl, dccp_send_synack,
272             econnp, NULL, SQ_PROCESS, SQTAG_TCP_SEND_SYNACK); /* XXX */
273     }
275     return;
276 error:
277 error2:
278 error3:
279     freemsg(mp);
280 }
282 void
283 dccp_input_listener_unbound(void *arg, mblk_t *mp, void *arg2,
284     ip_rcv_attr_t *ira)
285 {
286     conn_t      *connp = (conn_t *)arg;
287     queue_t     *sqp = (queue_t *)arg2;
288     queue_t     *new_sqp;
289     uint32_t    conn_flags;
291     cmn_err(CE_NOTE, "dccp_input.c: dccp_input_listener_unbound");
293     ASSERT(ira->ira_sqp != NULL);
294     new_sqp = ira->ira_sqp;
296     if (connp->conn_fanout == NULL) {
297         goto done;
298     }
300     /*
301     * Bind to correct queue.
302     */
303     if (!(connp->conn_flags & IPCL_FULLY_BOUND)) {
304         cmn_err(CE_NOTE, "not fully bound");
306         mutex_enter(&connp->conn_fanout->connf_lock);
307         mutex_enter(&connp->conn_lock);
309         if (connp->conn_ref != 4 ||
310             connp->conn_dccp->dccp_state != DCCPS_LISTEN) {
311             mutex_exit(&connp->conn_lock);
312             mutex_exit(&connp->conn_fanout->connf_lock);
313             goto done;
314         }
316         if (connp->conn_sqp != new_sqp) {
317             while (connp->conn_sqp != new_sqp) {
318                 (void) casptr(&connp->conn_sqp, sqp, new_sqp);
319             }
320             connp->conn_ixa->ixa_sqp = new_sqp;
321         }
323         do {
324             conn_flags = connp->conn_flags;
325             conn_flags |= IPCL_FULLY_BOUND;

```

```

326             (void) cas32(&connp->conn_flags, connp->conn_flags,
327                 conn_flags);
328         } while (!(connp->conn_flags & IPCL_FULLY_BOUND));
330         mutex_exit(&connp->conn_lock);
331         mutex_exit(&connp->conn_fanout->connf_lock);
333         connp->conn_rcv = dccp_input_listener;
334     }
336 done:
337     if (connp->conn_sqp != sqp) {
338         CONN_INC_REF(connp);
339         SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_rcv, connp,
340             ira, SQ_FILL, SQTAG_DCCP_CONN_REQ_UNBOUND);
341     } else {
342         dccp_input_listener(connp, mp, sqp, ira);
343     }
344 }
346 boolean_t
347 dccp_verifyicmp(conn_t *connp, void *arg2, icmph_t *icmph, icmp6_t *icmp6,
348     ip_rcv_attr_t *ira)
349 {
350     cmn_err(CE_NOTE, "dccp_input.c: dccp_verifyicmp");
352     return (B_TRUE);
353 }
354 #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp/dccp_ip.h

1

807 Wed Jun 13 12:04:34 2012

new/usr/src/uts/common/inet/dccp/dccp_ip.h

%B

```
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2012 David Hoepfner. All rights reserved.
14 */

16 #ifndef _INET_DCCP_DCCP_IP_H
17 #define _INET_DCCP_DCCP_IP_H

19 #include <netinet/dccp.h>
20 #include <inet/dccp/dccp_stack.h>

22 #ifdef __cplusplus
23 extern "C" {
24 #endif

26 /*
27  * DCCP functions for IP
28  */
29 extern void dccp_ddi_g_init(void);
30 extern void dccp_ddi_g_destroy(void);

33 #ifdef __cplusplus
34 }
35 #endif

37 #endif /* _INET_DCCP_DCCP_IP_H */
38 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_opt_data.c

1

```
*****
3415 Wed Jun 13 12:04:35 2012
new/usr/src/uts/common/inet/dccp/dccp_opt_data.c
%B
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */

27 /*
28  * Copyright 2012 David Hoepfner. All rights reserved.
29  */

31 /*
32  * This file contains functions related to getting and setting options
33  * thought the getsockopt and setsockopt socket functions.
34  */

36 #include <sys/types.h>
37 #include <sys/stream.h>
38 #define _SUN_TPI_VERSION 2
39 #include <sys/tihdr.h>
40 #include <sys/xti_xtiopt.h>
41 #include <sys/xti_inet.h>
42 #include <sys/policy.h>

44 #include <inet/common.h>
45 #include <inet/ip.h>
46 #include <inet/optcom.h>
47 #include <netinet/ip.h>

49 #include <sys/cmn_err.h>

51 #include "dccp_impl.h"

53 static int dccp_opt_default(queue_t *, int, int, uchar_t *);

55 /*
56  * Supported options.
57  */
58 opdes_t dccp_opt_arr[] = {
59 { SO_DEBUG, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
60 };
```

new/usr/src/uts/common/inet/dccp/dccp_opt_data.c

2

```
62 /*
63  * Supported levels.
64  */
65 optlevel_t dccp_valid_levels_arr[] = {
66     SOL_SOCKET,
67 };

69 #define DCCP_OPT_ARR_CNT      A_CNT(dccp_opt_arr)
70 #define DCCP_VALID_LEVELS_CNT A_CNT(dccp_valid_levels_arr)

72 uint_t dccp_max_optsize;

74 /*
75  * Options database object.
76  */
77 optdb_obj_t dccp_opt_obj = {
78     dccp_opt_default,
79     dccp_tpi_opt_get,
80     dccp_tpi_opt_set,
81     DCCP_OPT_ARR_CNT,
82     dccp_opt_arr,
83     DCCP_VALID_LEVELS_CNT,
84     dccp_valid_levels_arr,
85 };

87 /*
88  * Default value for certain options.
89  */
90 int
91 dccp_opt_default(queue_t *q, int level, int name, uchar_t *ptr)
92 {
93     dccp_stack_t *dccps = Q_TO_DCCP(q)->dccp_dccps;
94     int32_t *i1 = (int32_t *)ptr;

96     return (sizeof (int));
97 }

99 int
100 dccp_opt_get(conn_t *connp, int level, int name, uchar_t *ptr)
101 {
102     dccp_t *dccp = connp->conn_dccp;
103     conn_opt_arg_t coas;
104     int retval;

106     coas.coa_connp = connp;
107     coas.coa_ixa = connp->conn_ixa;
108     coas.coa_ipp = &connp->conn_xmit_ipp;
109     coas.coa_ancillary = B_FALSE;
110     coas.coa_changed = 0;

112     switch (level) {
113     case SOL_SOCKET:
114         break;
115     case IPPROTO_TCP:
116         break;
117     case IPPROTO_IP:
118         break;
119     case IPPROTO_IPV6:
120         break;
121     }

123     mutex_enter(&connp->conn_lock);
124     retval = conn_opt_get(&coas, level, name, ptr);
125     mutex_exit(&connp->conn_lock);

127     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;
141
142     coas.coa_connp = connp;
143     coas.coa_ancillary = B_FALSE;
144     coas.coa_changed = 0;
145
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     }
152
153     return (0);
154 }
155 #endif /* ! codereview */
```



```

*****
13439 Wed Jun 13 12:04:36 2012
new/usr/src/uts/common/inet/dccp/dccp_socket.c
%B
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */

27 /*
28  * Copyright 2012 David Hoepfner. All rights reserved.
29  */

31 /*
32  * This file contains function related to the socket interface.
33  */

35 #include <sys/types.h>
36 #include <sys/strlog.h>
37 #include <sys/policy.h>
38 #include <sys/sockio.h>
39 #include <sys/strsubr.h>
40 #include <sys/strsun.h>
41 #define _SUN_TPI_VERSION 2
42 #include <sys/tihdr.h>
43 #include <sys/squeue_impl.h>
44 #include <sys/squeue.h>
45 #include <sys/socketvar.h>

47 #include <inet/common.h>
48 #include <inet/proto_set.h>
49 #include <inet/ip.h>

51 #include <sys/cmn_err.h>

53 #include "dccp_impl.h"
54 #include "dccp_stack.h"

56 static void    dccp_activate(sock_lower_handle_t, sock_upper_handle_t,
57                          sock_upcalls_t *, int, cred_t *);
58 static int     dccp_accept(sock_lower_handle_t, sock_lower_handle_t,
59                          sock_upper_handle_t, cred_t *);
60 static int     dccp_bind(sock_lower_handle_t, struct sockaddr *,
61                          socklen_t, cred_t *);

```

```

62 static int     dccp_listen(sock_lower_handle_t, int, cred_t *);
63 static int     dccp_connect(sock_lower_handle_t, const struct sockaddr *,
64                          socklen_t, sock_connid_t *, cred_t *);
65 static int     dccp_getpeername(sock_lower_handle_t, struct sockaddr *,
66                          socklen_t *, cred_t *);
67 static int     dccp_getsockname(sock_lower_handle_t, struct sockaddr *,
68                          socklen_t *, cred_t *);
69 static int     dccp_getsockopt(sock_lower_handle_t, int, int, void *,
70                          socklen_t *, cred_t *);
71 static int     dccp_setsockopt(sock_lower_handle_t, int, int, const void *,
72                          socklen_t, cred_t *);
73 static int     dccp_sendmsg(sock_lower_handle_t, mblk_t *, struct nmsg_hdr *,
74                          cred_t *);
75 static int     dccp_shutdown(sock_lower_handle_t, int, cred_t *);
76 static void    dccp_clr_flowctrl(sock_lower_handle_t);
77 static int     dccp_ioctl(sock_lower_handle_t, int, intptr_t, int, int32_t *,
78                          cred_t *);
79 static int     dccp_close(sock_lower_handle_t, int, cred_t *);

81 sock_downcalls_t sock_dccp_downcalls = {
82     dccp_activate,
83     dccp_accept,
84     dccp_bind,
85     dccp_listen,
86     dccp_connect,
87     dccp_getpeername,
88     dccp_getsockname,
89     dccp_getsockopt,
90     dccp_setsockopt,
91     dccp_sendmsg,
92     NULL,
93     NULL,
94     NULL,
95     dccp_shutdown,
96     dccp_clr_flowctrl,
97     dccp_ioctl,
98     dccp_close,
99 };

101 /* ARGSUSED */
102 static void
103 dccp_activate(sock_lower_handle_t proto_handle, sock_upper_handle_t sock_handle,
104             sock_upcalls_t *sock_upcalls, int flags, cred_t *cr)
105 {
106     conn_t *connp = (conn_t *)proto_handle;
107     struct sock_proto_props    sopp;
108     //extern struct module_info    tcp_rinfo;

110     cmn_err(CE_NOTE, "dccp_socket.c: dccp_activate");

112     ASSERT(cr != NULL);

114     sopp.sopp_flags = SOCKOPT_RCVHIWAT | SOCKOPT_RCVLOWAT |
115                   SOCKOPT_MAXPSZ | SOCKOPT_MAXBLK | SOCKOPT_RCVTIMER |
116                   SOCKOPT_RCVTHRESH | SOCKOPT_MAXADDRLEN | SOCKOPT_MINPSZ;

119     sopp.sopp_rxhiwat = SOCKET_RECVHIWATER;
120     sopp.sopp_rxlowat = SOCKET_RECVLOWATER;
121     sopp.sopp_maxpsz = INFPSZ;
122     sopp.sopp_maxblk = INFPSZ;
123     sopp.sopp_rcvtimer = SOCKET_TIMER_INTERVAL;
124     sopp.sopp_rcvthresh = SOCKET_RECVHIWATER >> 3;
125     sopp.sopp_maxaddrlen = sizeof (sin6_t);
126 /*
127     sopp.sopp_minpsz = (dccp_rinfo.mi_minpsz == 1) ? 0 :

```



```

128     dccp_rinfo.mi_minpsz;
129 */
130     connp->conn_upcalls = sock_upcalls;
131     connp->conn_upper_handle = sock_handle;
132
133     /* XXX */
134     (*connp->conn_upcalls->su_set_proto_props)(connp->conn_upper_handle,
135     &sockp);
136 }
137
138 /*ARGSUSED*/
139 static int
140 dccp_accept(sock_lower_handle_t lproto_handle,
141 sock_lower_handle_t eproto_handle, sock_upper_handle_t sock_handle,
142 cred_t *cr)
143 {
144     cmn_err(CE_NOTE, "dccp_socket.c: dccp_accept");
145
146     return (ENOTSUP);
147 }
148
149 static int
150 dccp_bind(sock_lower_handle_t proto_handle, struct sockaddr *sa,
151 socklen_t len, cred_t *cr)
152 {
153     conn_t *connp = (conn_t *)proto_handle;
154     int error;
155
156     cmn_err(CE_NOTE, "dccp_socket.c: dccp_bind");
157
158     ASSERT(cr != NULL);
159     ASSERT(connp->conn_upper_handle != NULL);
160
161     error = squeue_synch_enter(connp, NULL);
162     if (error != 0) {
163         /* Failed to enter */
164         return (ENOSR);
165     }
166
167     /* Binding to NULL address means unbind */
168     if (sa == NULL) {
169         if (connp->conn_dccp->dccp_state < DCCPS_LISTEN) {
170             error = dccp_do_unbind(connp);
171         } else {
172             error = EINVAL;
173         }
174     } else {
175         error = dccp_do_bind(connp, sa, len, cr, B_TRUE);
176     }
177
178     squeue_synch_exit(connp);
179
180     if (error < 0) {
181         if (error == -TOUTSTATE) {
182             error = EINVAL;
183         } else {
184             error = proto_tlitossyserr(-error);
185         }
186     }
187
188     return (error);
189 }
190
191 /* ARGSUSED */
192 static int
193 dccp_listen(sock_lower_handle_t proto_handle, int backlog, cred_t *cr)

```

```

194 {
195     conn_t *connp = (conn_t *)proto_handle;
196     dccp_t *dccp = connp->conn_dccp;
197     int error;
198
199     cmn_err(CE_NOTE, "dccp_socket.c: dccp_listen");
200
201     ASSERT(connp->conn_upper_handle != NULL);
202     ASSERT(cr != NULL);
203
204     error = squeue_synch_enter(connp, NULL);
205     if (error != 0) {
206         /* Failed to enter */
207         return (ENOBUFS);
208     }
209
210     error = dccp_do_listen(connp, NULL, 0, backlog, cr, B_FALSE);
211     if (error == 0) {
212         /* XXX:DCCP */
213         (*connp->conn_upcalls->su_opctl)(connp->conn_upper_handle,
214         SOCK_OPCTL_ENAB_ACCEPT,
215         (uintptr_t)(10));
216     } else if (error < 0) {
217         if (error == -TOUTSTATE) {
218             error = EINVAL;
219         } else {
220             error = proto_tlitossyserr(-error);
221         }
222     }
223
224     squeue_synch_exit(connp);
225
226     return (error);
227 }
228
229 static int
230 dccp_connect(sock_lower_handle_t proto_handle, const struct sockaddr *sa,
231 socklen_t len, sock_connid_t *id, cred_t *cr)
232 {
233     conn_t *connp = (conn_t *)proto_handle;
234     int error;
235
236     cmn_err(CE_NOTE, "dccp_socket.c: dccp_connect");
237
238     ASSERT(connp->conn_upper_handle != NULL);
239     ASSERT(cr != NULL);
240
241     error = proto_verify_ip_addr(connp->conn_family, sa, len);
242     if (error != 0) {
243         return (error);
244     }
245
246     error = squeue_synch_enter(connp, NULL);
247     if (error != 0) {
248         /* Failed to enter */
249         return (ENOSR);
250     }
251
252     error = dccp_do_connect(connp, sa, len, cr, curproc->p_pid);
253     if (error == 0) {
254         *id = connp->conn_dccp->dccp_connid;
255     } else if (error < 0) {
256         if (error == -TOUTSTATE) {
257             switch (connp->conn_dccp->dccp_state) {
258                 /* XXX */
259                 case DCCPS_LISTEN:

```

```

260         error = EOPNOTSUPP;
261         break;
262     default:
263         error = EINVAL;
264         break;
265     } else {
266         error = proto_tlitossyserr(-error);
267     }
268 }
269
271     squeue_synch_exit(connp);
272
273     cmn_err(CE_NOTE, "dccp_connect.c: exit %d", error);
274     return ((error == 0) ? EINPROGRESS : error);
275 }
276
277 /* ARGSUSED3 */
278 static int
279 dccp_getpeername(sock_lower_handle_t proto_handle, struct sockaddr *addr,
280                 socklen_t *addrlenp, cred_t *cr)
281 {
282     conn_t *connp = (conn_t *)proto_handle;
283     dccp_t *dccp = connp->conn_dccp;
284
285     cmn_err(CE_NOTE, "dccp_socket.c: dccp_getpeername");
286
287     ASSERT(cr != NULL);
288
289     ASSERT(dccp != NULL);
290     /* XXX:DCCP */
291
292     return (conn_getpeername(connp, addr, addrlenp));
293 }
294
295 /* ARGSUSED3 */
296 static int
297 dccp_getsockname(sock_lower_handle_t proto_handle, struct sockaddr *addr,
298                 socklen_t *addrlenp, cred_t *cr)
299 {
300     conn_t *connp = (conn_t *)proto_handle;
301     int error;
302
303     cmn_err(CE_NOTE, "dccp_socket.c: dccp_getsockname");
304
305     ASSERT(cr != NULL);
306
307     /* XXX UDP has locks here, TCP not */
308     mutex_enter(&connp->conn_lock);
309     error = conn_getsockname(connp, addr, addrlenp);
310     mutex_exit(&connp->conn_lock);
311
312     return (error);
313 }
314
315 static int
316 dccp_getsockopt(sock_lower_handle_t proto_handle, int level, int option_name,
317                void *optvalp, socklen_t *optlen, cred_t *cr)
318 {
319     conn_t *connp = (conn_t *)proto_handle;
320     t_uscalar_t max_optbuf_len;
321     void *optvalp_buf;
322     int len;
323     int error;
324
325     cmn_err(CE_NOTE, "dccp_socket.c: dccp_getsockopt");

```

```

327     ASSERT(connp->conn_upper_handle != NULL);
328
329     error = proto_opt_check(level, option_name, *optlen, &max_optbuf_len,
330                             dccp_opt_obj.odb_opt_des_arr,
331                             dccp_opt_obj.odb_opt_arr_cnt,
332                             B_FALSE, B_TRUE, cr);
333     if (error != 0) {
334         if (error < 0) {
335             error = proto_tlitossyserr(-error);
336         }
337         return (error);
338     }
339
340     optvalp_buf = kmem_alloc(max_optbuf_len, KM_SLEEP);
341     if (optvalp_buf == NULL) {
342         return (ENOMEM);
343     }
344
345     error = squeue_synch_enter(connp, NULL);
346     if (error == ENOMEM) {
347         kmem_free(optvalp_buf, max_optbuf_len);
348         return (ENOMEM);
349     }
350
351     len = dccp_opt_get(connp, level, option_name, optvalp_buf);
352     squeue_synch_exit(connp);
353
354     if (len == -1) {
355         kmem_free(optvalp_buf, max_optbuf_len);
356         return (EINVAL);
357     }
358
359     t_uscalar_t size = MIN(len, *optlen);
360
361     bcopy(optvalp_buf, optvalp, size);
362     bcopy(&size, optlen, sizeof(size));
363
364     kmem_free(optvalp_buf, max_optbuf_len);
365
366     return (0);
367 }
368
369 static int
370 dccp_setsockopt(sock_lower_handle_t proto_handle, int level, int option_name,
371                const void *optvalp, socklen_t optlen, cred_t *cr)
372 {
373     conn_t *connp = (conn_t *)proto_handle;
374     int error;
375
376     cmn_err(CE_NOTE, "dccp_socket.c: dccp_setsockopt");
377
378     ASSERT(connp->conn_upper_handle != NULL);
379
380     error = squeue_synch_enter(connp, NULL);
381     if (error == ENOMEM) {
382         return (ENOMEM);
383     }
384
385     error = proto_opt_check(level, option_name, optlen, NULL,
386                             dccp_opt_obj.odb_opt_des_arr,
387                             dccp_opt_obj.odb_opt_arr_cnt,
388                             B_TRUE, B_FALSE, cr);
389     if (error != 0) {
390         if (error < 0) {
391             error = proto_tlitossyserr(-error);

```

```

392     }
393     squeue_synch_exit(connp);
394     return (error);
395 }

397 error = dccp_opt_set(connp, SETFN_OPTCOM_NEGOTIATE, level, option_name,
398 optlen, (uchar_t *)optvalp, (uint_t *)&optlen, (uchar_t *)optvalp,
399 NULL, cr);
400 squeue_synch_exit(connp);

402 ASSERT(error >= 0);

404 return (error);
405 }

407 /* ARGSUSED */
408 static int
409 dccp_sendmsg(sock_lower_handle_t proto_handle, mblk_t *mp, struct nmsg_hdr *msg,
410 cred_t *cr)
411 {
412     conn_t *connp = (conn_t *)proto_handle;
413     dccp_t *dccp;
414     uint32_t msize;
415     int32_t dccpstate;

417     cmn_err(CE_NOTE, "dccp_socket.c: dccp_sendmsg");

419     /* All Solaris components should pass a cred for this operation. */
420     ASSERT(cr != NULL);

422     ASSERT(connp->conn_ref >= 2);
423     ASSERT(connp->conn_upper_handle != NULL);

425     if (msg->msg_controllen != 0) {
426         freemsg(mp);
427         return (EOPNOTSUPP);
428     }

430     switch (DE_TYPE(mp)) {
431     case M_DATA:
432         dccp = connp->conn_dccp;
433         ASSERT(dccp != NULL);

435         dccpstate = dccp->dccp_state;

437         /* XXX */

439         return (0);

441     default:
442         ASSERT(0);
443     }

445     freemsg(mp);

447     return (0);
448 }

450 /* ARGSUSED */
451 static int
452 dccp_shutdown(sock_lower_handle_t proto_handle, int how, cred_t *cr)
453 {
454     conn_t *connp = (conn_t *)proto_handle;
455     dccp_t *dccp = connp->conn_dccp;

457     cmn_err(CE_NOTE, "dccp_socket.c: dccp_shutdown");

```

```

459     /* All Solaris components should pass a cred for this operation. */
460     ASSERT(cr != NULL);

462     ASSERT(connp->conn_upper_handle != NULL);

465     return (ENOTSUP);
466 }

468 static void
469 dccp_clr_flowctrl(sock_lower_handle_t proto_handle)
470 {
471     conn_t *connp = (conn_t *)proto_handle;
472     dccp_t *dccp = connp->conn_dccp;
473     mblk_t *mp;
474     int error;

476     ASSERT(connp->conn_upper_handle != NULL);

478     cmn_err(CE_NOTE, "dccp_socket.c: dccp_clr_flowctrl");

480     error = squeue_synch_enter(connp, mp);

482     squeue_synch_exit(connp);
483 }

485 /* ARGSUSED */
486 static int
487 dccp_ioctl(sock_lower_handle_t proto_handle, int cmd, intptr_t arg,
488 int mode, int32_t *rvalp, cred_t *cr)
489 {
490     conn_t *connp = (conn_t *)proto_handle;
491     int error;

493     cmn_err(CE_NOTE, "dccp_socket.c: dccp_ioctl");

495     ASSERT(connp->conn_upper_handle != NULL);

497     /* All Solaris components should pass a cred for this operation. */
498     ASSERT(cr != NULL);

500     return (ENOTSUP);
501 }

503 /* ARGSUSED */
504 static int
505 dccp_close(sock_lower_handle_t proto_handle, int flags, cred_t *cr)
506 {
507     conn_t *connp = (conn_t *)proto_handle;

509     cmn_err(CE_NOTE, "dccp_socket.c: dccp_close\n");

511     ASSERT(connp->conn_upper_handle != NULL);

513     /* All Solaris components should pass a cred for this operation. */

515     ASSERT(cr != NULL);

517     dccp_close_common(connp);

519     ip_free_helper_stream(connp);

521     CONN_DEC_REF(connp);

523     return (EINPROGRESS);

```

```
524 }

527 /*
528  * Socket create function.
529  */
530 sock_lower_handle_t
531 dccp_create(int family, int type, int proto, sock_downcalls_t **sockdowncalls,
532            uint_t *smodep, int *errorp, int flags, cred_t *credp)
533 {
534     conn_t      *connp;
535     boolean_t   isv6;

537     /* XXX (type != SOCK_STREAM */
538     if ((family != AF_INET && family != AF_INET6) ||
539         (proto != 0 && proto != IPPROTO_DCCP)) {
540         *errorp = EPROTONOSUPPORT;
541         return (NULL);
542     }

544     cmn_err(CE_NOTE, "dccp_socket: dccp_create\n");

546     isv6 = family == AF_INET6 ? B_TRUE: B_FALSE;
547     connp = dccp_create_common(credp, isv6, B_TRUE, errorp);
548     if (connp == NULL) {
549         return (NULL);
550     }

552     /*
553      * Increment ref for DCCP connection.
554      */
555     mutex_enter(&connp->conn_lock);
556     CONN_INC_REF_LOCKED(connp);
557     ASSERT(connp->conn_ref == 2);
558     connp->conn_state_flags &= ~CONN_INCIPIENT;
559     connp->conn_flags |= IPCL_NONSTR;
560     mutex_exit(&connp->conn_lock);

562     ASSERT(errorp != NULL);
563     *errorp = 0;
564     *sockdowncalls = &sock_dccp_downcalls;
565     *smodep = SM_CONNREQUIRED | SM_EXDATA | SM_ACCEPTSUPP |
566             SM_SENDFILESUPP;

568     return ((sock_lower_handle_t)connp);
569 }

571 int
572 dccp_fallback(sock_lower_handle_t proto_handle, queue_t *q,
573              boolean_t issocket, so_proto_quiesced_cb_t quiesced_cb,
574              sock_quiesce_arg_t *arg)
575 {
576     cmn_err(CE_NOTE, "dccp_socket: dccp_fallback\n");

578     return (0);
579 }
580 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_stack.h

1

```
*****
1677 Wed Jun 13 12:04:40 2012
new/usr/src/uts/common/inet/dccp/dccp_stack.h
%B
*****
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2012 David Hoepfner. All rights reserved.
14 */

16 #ifndef _INET_DCCP_DCCP_STACK_H
17 #define _INET_DCCP_DCCP_STACK_H

19 #include <sys/netstack.h>
20 #include <sys/cpuvar.h>

22 #ifdef __cplusplus
23 extern "C" {
24 #endif

26 typedef struct dccp_stat_counter_s {
27     uint64_t      dccp_rst_unsent;
28 } dccp_stat_counter_t;

30 typedef struct {
31     uint64_t      dccp_stats_cnt;
32     dccp_stat_counter_t  dccp_sc_stats;
33 } dccp_stats_cpu_t;

35 #define DCCP_STAT(dccps, x) \
36     ((dccps)->dccps_sc[CPU->cpu_seqid]->dccp_sc_stats.x++)

38 /*
39  * DCCP stack instances
40  */
41 typedef struct dccp_stack {
42     netstack_t      *dccps_netstack;      /* Common netstack */
44     uint_t           dccps_bind_fanout_size;
45     struct dccp_df_s *dccps_bind_fanout;

47     /* Ports */
48 #define DCCP_NUM_EPRIV_PORTS 64
49     int             dccps_num_epriv_ports;
50     dccps_epriv_ports[DCCP_NUM_EPRIV_PORTS];
51     kmutex_t        dccps_epriv_port_lock;

53     uint_t           dccps_next_port_to_try;

55     in_port_t        dccps_min_anonpriv_port;

57     /* Reset rate control */
58     int64_t          dccps_last_rst_intrvl;
59     uint32_t         dccps_rst_cnt;

61     /* Tunables table */
```

new/usr/src/uts/common/inet/dccp/dccp_stack.h

2

```
62     struct mod_prop_info_s *dccps_propinfo_tbl;

64     ldi_ident_t      dccps_ldi_ident;

66     /* Cpu stats counter */
67     dccp_stats_cpu_t **dccps_sc;
68     int             dccps_sc_cnt;
69 } dccp_stack_t;

71 #ifdef __cplusplus
72 }
73 #endif

75 #endif /* _INET_DCCP_DCCP_STACK_H */
76 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_stats.c

1

```
*****
3624 Wed Jun 13 12:04:40 2012
```

new/usr/src/uts/common/inet/dccp/dccp_stats.c

```
%B
*****
```

```
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 */
21
22 /*
23 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */
26
27 /*
28 * Copyright 2012 David Hoepfner. All rights reserved.
29 */
30
31 /*
32 * Functions related to MIB-II and kstat.
33 */
34
35 #include <sys/types.h>
36 #include <sys/tihdr.h>
37 #include <sys/policy.h>
38 #include <sys/tsol/tnet.h>
39
40 #include <inet/common.h>
41 #include <inet/ip.h>
42 #include <inet/kstatcom.h>
43 #include <inet/snmpcom.h>
44
45 #include <sys/cmn_err.h>
46
47 #include "dccp_impl.h"
48
49 static int dccp_snmp_state(dccp_t *);
50
51 static int
52 dccp_snmp_state(dccp_t *dccp)
53 {
54     if (dccp == NULL) {
55         return (0);
56     }
57
58     switch(dccp->dccp_state) {
59     case DCCPS_CLOSED:
60         return (MIB2_DCCP_closed);
61     default:
```

new/usr/src/uts/common/inet/dccp/dccp_stats.c

2

```
62         return (0);
63     }
64 }
65
66 /*
67  * Get the MIB-II stats.
68  */
69 mblk_t *
70 dccp_snmp_get(queue_t *q, mblk_t *mpctl)
71 {
72     conn_t          *connp = Q_TO_CONN(q);
73     connf_t         *connfp;
74     ip_stack_t      *ips;
75     dccp_stack_t    *dccps;
76     struct ophdr    *optp;
77     mblk_t          *mp2ctl;
78     mblk_t          *mpdata;
79     mblk_t          *mp_conn_ctl = NULL;
80     mblk_t          *mp_conn_tail;
81     mblk_t          *mp_attr_ctl = NULL;
82     mblk_t          *mp_attr_tail;
83     size_t          dccp_mib_size;
84     size_t          dce_size;
85     zoneid_t        zoneid;
86     int             i;
87     mib2_dccpConnEntry_t dce;
88
89     mp2ctl = copymsg(mpctl);
90
91     if (mpctl == NULL ||
92         (mpdata = mpctl->b_cont) == NULL ||
93         (mp_conn_ctl = copymsg(mpctl)) == NULL ||
94         (mp_attr_ctl = copymsg(mpctl)) == NULL) {
95         freemsg(mp_conn_ctl);
96         freemsg(mp_attr_ctl);
97         freemsg(mpctl);
98         freemsg(mp2ctl);
99         return (NULL);
100    }
101
102    ips = connp->conn_netstack->netstack_ip;
103    dccps = connp->conn_netstack->netstack_dccp;
104    dce_size = sizeof (mib2_dccpConnEntry_t);
105    zoneid = Q_TO_CONN(q)->conn_zoneid;
106
107    for (i = 0; i < CONN_G_HASH_SIZE; i++) {
108        ips = dccps->dccps_netstack->netstack_ip;
109
110        connfp = &ips->ipcl_globalhash_fanout[i];
111        connp = NULL;
112
113        while ((connp = ipcl_get_next_conn(connfp, connp,
114            IPCL_DCCPCONN)) != NULL) {
115            dccp_t *dccp;
116
117            if (connp->conn_zoneid != zoneid) {
118                continue;
119            }
120
121            dccp = connp->conn_dccp;
122
123            dce.dccpConnState = dccp_snmp_state(dccp);
124
125            if (connp->conn_ipversion == IPV4_VERSION ||
126                (dccp->dccp_state <= DCCPS_LISTEN)) {
127                dce.dccpConnRemAddress =
```

```
128         connp->conn_faddr_v4;
129         dce.dccpConnLocalAddress =
130         connp->conn_laddr_v4;
131     }
132
133     dce.dccpConnLocalPort = ntohs(connp->conn_lport);
134     dce.dccpConnRemPort = ntohs(connp->conn_fport);
135
136     dce.dccpConnCreationProcess = (connp->conn_cpid < 0) ?
137     MIB2_UNKNOWN_PROCESS : connp->conn_cpid;
138     dce.dccpConnCreationTime = connp->conn_open_time;
139
140     (void) snmp_append_data2(mp_conn_ctl->b_cont,
141     &mp_conn_tail, (char *)&dce, dce_size);
142 }
143
144
145     optp = (struct ophdr *)&mp_conn_ctl->b_rptr[
146     sizeof (struct T_optmgmt_ack)];
147     optp->level = MIB2_DCCP;
148     optp->name = MIB2_DCCP_CONN;
149     optp->len = msgdsize(mp_conn_ctl->b_cont);
150     qreply(q, mp_conn_ctl);
151
152     return (mp2ctl);
153 }
154 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_tpi.c

1

```
*****
2671 Wed Jun 13 12:04:41 2012
new/usr/src/uts/common/inet/dccp/dccp_tpi.c
%B
*****
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  * Functions related to TPI.
24  */

26 #include <sys/types.h>
27 #include <sys/stream.h>
28 #include <sys/strsun.h>
29 #include <sys/strsubr.h>
30 #include <sys/stropts.h>
31 #include <sys/strlog.h>
32 #define _SUN_TPI_VERSION 2
33 #include <sys/tihdr.h>
34 #include <sys/suntpi.h>
35 #include <sys/xti_inet.h>
36 #include <sys/squeue_impl.h>
37 #include <sys/squeue.h>
38 #include <sys/tsol/tnet.h>

40 #include <inet/common.h>
41 #include <inet/ip.h>

43 #include <sys/cmn_err.h>

45 #include "dccp_impl.h"

47 /*
48  * Helper function to generate TPI errors acks.
49  */
50 void
51 dccp_err_ack(dccp_t *dccp, mblk_t *mp, int t_error, int sys_error)
52 {
53     if ((mp = mi_tpi_err_ack_alloc(mp, t_error, sys_error)) != NULL) {
54         putnext(dccp->dccp_connp->conn_rq, mp);
55     }
56 }

58 void
59 dccp_tpi_connect(dccp_t *dccp, mblk_t *mp)
60 {
61     struct T_conn_req    *tcr;
```

new/usr/src/uts/common/inet/dccp/dccp_tpi.c

2

```
62     conn_t    *connp = dccp->dccp_connp;
63     sin_t     *sin;
64     sin6_t    *sin6;
65     cred_t    *cr;
66     pid_t     cpid;
67     int       error;

69     cmn_err(CE_NOTE, "dccp_tpi.c: dccp_tpi_connect");

71     cr = msg_getcred(mp, &cpid);
72     ASSERT(cr != NULL);
73     if (cr == NULL) {
74         dccp_err_ack(dccp, mp, TSYSEERR, EINVAL);
75         return;
76     }

78     tcr = (struct T_conn_req *)mp->b_rptr;

80     ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <= (uintptr_t)INT_MAX);
81     if ((mp->b_wptr - mp->b_rptr) < sizeof (*tcr)) {
82         dccp_err_ack(dccp, mp, TPROTO, 0);
83         return;
84     }

86 }

88 int
89 dccp_tpi_close(queue_t *q, int flags)
90 {
91     return (0);
92 }

94 /*
95  * Options related functions.
96  */
97 int
98 dccp_tpi_opt_get(queue_t *q, int level, int name, uchar_t *ptr)
99 {
100     return (tcp_opt_get(Q_TO_CONN(q), level, name, ptr));
101 }

103 /* ARGSUSED */
104 int
105 dccp_tpi_opt_set(queue_t *q, uint_t optset_context, int level, int name,
106     uint_t inlen, uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
107     void *thisdg_attrs, cred_t *cr)
108 {
109     conn_t    *connp = Q_TO_CONN(q);

111     return (tcp_opt_set(connp, optset_context, level, name, inlen, invalp,
112         outlenp, outvalp, thisdg_attrs, cr));
113 }
114 #endif /* ! codereview */
```


new/usr/src/uts/common/inet/dccp/dccp_tunables.c

1

1647 Wed Jun 13 12:04:42 2012

new/usr/src/uts/common/inet/dccp/dccp_tunables.c

%B

```
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2012 David Hoepfner. All rights reserved.
14 */

16 /*
17  * This file contains tunable properties for DCCP.
18 */
19 #include <inet/ip.h>
20 #include <inet/ip6.h>
21 #include <inet/dccp/dccp_impl.h>
22 #include <sys/sunddi.h>

24 mod_prop_info_t dccp_propinfo_tbl[] = {
25     /* tunable - 0 */
26     { "smallest_nonpriv_port", MOD_PROTO_DCCP,
27       mod_set_uint32, mod_get_uint32,
28       {1024, (32 * 1024), 1024}, {1024} },

30     { "smallest_anon_port", MOD_PROTO_DCCP,
31       mod_set_uint32, mod_get_uint32,
32       {1024, ULP_MAX_PORT, 32*1024}, {32*1024} },

34     { "largest_anon_port", MOD_PROTO_DCCP,
35       mod_set_uint32, mod_get_uint32,
36       {1024, ULP_MAX_PORT, ULP_MAX_PORT}, {ULP_MAX_PORT} },

38     { "_xmit_lowat", MOD_PROTO_DCCP,
39       mod_set_uint32, mod_get_uint32,
40       {0, (1<<30), DCCP_XMIT_LOWATER},
41       {DCCP_XMIT_LOWATER} },

43     { "_debug", MOD_PROTO_DCCP,
44       mod_set_uint32, mod_get_uint32,
45       {0, 10, 0}, {0} },

47     { "_rst_sent_rate_enabled", MOD_PROTO_DCCP,
48       mod_set_boolean, mod_get_boolean,
49       {B_TRUE}, {B_TRUE} },

51     { "_rst_sent_rate", MOD_PROTO_DCCP,
52       mod_set_uint32, mod_get_uint32,
53       {0, UINT32_MAX, 40}, {40} },

55     /* tunable - 10 */

57     { NULL, 0, NULL, NULL, {0}, {0} }
58 };

60 int dccp_propinfo_count = A_CNT(dccp_propinfo_tbl);
61 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccpddi.c

1

1439 Wed Jun 13 12:04:42 2012

new/usr/src/uts/common/inet/dccp/dccpddi.c

%B

```
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2012 David Hoepfner. All rights reserved.
14 */

16 #include <sys/types.h>
17 #include <sys/conf.h>
18 #include <sys/modctl.h>
19 #include <inet/common.h>
20 #include <inet/ip.h>
21 #include <sys/strsubr.h>
22 #include <sys/socketvar.h>

24 #include "dccp_impl.h"

26 #define INET_NAME "dccp"
27 #define INET_MODDESC "DCCP dummy STREAMS module"
28 #define INET_DEVDESC "DCCP STREAMS driver"
29 #define INET_SOCKDESC "DCCP socket module"
30 #define INET_MODSTRTAB dummymodinfo
31 #define INET_DEVSTRTAB dccpinfov4
32 #define INET_MODMTFLAGS D_MP
33 #define INET_SOCK_PROTO_CREATE_FUNC (*dccp_create)
34 #define INET_SOCK_PROTO_FB_FUNC (*dccp_fallback)
35 #define INET_SOCK_FALLBACK_DEV_V4 "/dev/dccp"
36 #define INET_SOCK_FALLBACK_DEV_V6 "/dev/dccp6"
37 #define INET_DEVMINOR 0
38 #define INET_MODMTFLAGS D_MP
39 #define INET_DEVMTFLAGS (D_MP|_D_DIRECT)

41 #include "../inetddi.c"

43 int
44 _init(void)
45 {
46     return (mod_install(&modlinkage));
47 }

49 int
50 _fini(void)
51 {
52     return (mod_remove(&modlinkage));
53 }

55 int
56 _info(struct modinfo *modinfop)
57 {
58     return (mod_info(&modlinkage, modinfop));
59 }
60 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/ip.h

1

```
*****
140153 Wed Jun 13 12:04:43 2012
new/usr/src/uts/common/inet/ip.h
%B
*****
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/queue.h>
55 #include <net/route.h>
56 #include <sys/system.h>
57 #include <net/radix.h>
58 #include <sys/modhash.h>

60 #ifdef DEBUG
61 #define CONN_DEBUG
```

new/usr/src/uts/common/inet/ip.h

2

```
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        IPV4_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
```

```

128 * 2 files should be cleaned up to remove all redundant definitions.
129 */
130 #define IP_MAXPACKET                65535
131 #define IP_SIMPLE_HDR_VERSION \
132     ((IP_VERSION << 4) | IP_SIMPLE_HDR_LENGTH_IN_WORDS)

134 #define UDPH_SIZE                    8

136 /*
137 * Constants and type definitions to support IP IOCTL commands
138 */
139 #define IP_IOCTL                    (('i'<<8)|'p')
140 #define IP_IOC_IRE_DELETE           4
141 #define IP_IOC_IRE_DELETE_NO_REPLY 5
142 #define IP_IOC_RTS_REQUEST          7

144 /* Common definitions used by IP IOCTL data structures */
145 typedef struct ipllcmd_s {
146     uint_t ipllc_cmd;
147     uint_t ipllc_name_offset;
148     uint_t ipllc_name_length;
149 } ipllc_t;

151 /* IP IRE Delete Command Structure. */
152 typedef struct ipid_s {
153     ipllc_t ipid_ipllc;
154     uint_t ipid_ire_type;
155     uint_t ipid_addr_offset;
156     uint_t ipid_addr_length;
157     uint_t ipid_mask_offset;
158     uint_t ipid_mask_length;
159 } ipid_t;

161 #define ipid_cmd                    ipid_ipllc.ipllc_cmd

163 #ifdef _KERNEL
164 /*
165 * Temporary state for ip options parser.
166 */
167 typedef struct ipoptp_s
168 {
169     uint8_t *ipoptp_next; /* next option to look at */
170     uint8_t *ipoptp_end; /* end of options */
171     uint8_t *ipoptp_cur; /* start of current option */
172     uint8_t ipoptp_len; /* length of current option */
173     uint32_t ipoptp_flags;
174 } ipoptp_t;

176 /*
177 * Flag(s) for ipoptp_flags
178 */
179 #define IPOPTP_ERROR 0x00000001
180 #endif /* _KERNEL */

182 /* Controls forwarding of IP packets, set via ipadm(1M)/ndd(1M) */
183 #define IP_FORWARD_NEVER 0
184 #define IP_FORWARD_ALWAYS 1

186 #define WE_ARE_FORWARDING(ipst) ((ipst)->ips_ip_forwarding == IP_FORWARD_ALWAYS)

188 #define IPH_HDR_LENGTH(ipha) \
189     ((int)((ipha_t *)ipha)->ipha_version_and_hdr_length & 0xF) << 2)

191 #define IPH_HDR_VERSION(ipha) \
192     ((int)((ipha_t *)ipha)->ipha_version_and_hdr_length) >> 4)

```

```

194 #ifdef _KERNEL
195 /*
196 * IP reassembly macros. We hide starting and ending offsets in b_next and
197 * b_prev of messages on the reassembly queue. The messages are chained using
198 * b_cont. These macros are used in ip_reassemble() so we don't have to see
199 * the ugly casts and assignments.
200 * Note that the offsets are <= 64k i.e. a uint_t is sufficient to represent
201 * them.
202 */
203 #define IP_REASS_START(mp)          ((uint_t)(uintptr_t)((mp)->b_next))
204 #define IP_REASS_SET_START(mp, u)  \
205     ((mp)->b_next = (mblk_t *) (uintptr_t)(u))
206 #define IP_REASS_END(mp)          ((uint_t)(uintptr_t)((mp)->b_prev))
207 #define IP_REASS_SET_END(mp, u)   \
208     ((mp)->b_prev = (mblk_t *) (uintptr_t)(u))

210 #define IP_REASS_COMPLETE          0x1
211 #define IP_REASS_PARTIAL          0x2
212 #define IP_REASS_FAILED           0x4

214 /*
215 * Test to determine whether this is a module instance of IP or a
216 * driver instance of IP.
217 */
218 #define CONN_Q(q)                  (WR(q)->q_next == NULL)

220 #define Q_TO_CONN(q)              ((conn_t *) (q)->q_ptr)
221 #define Q_TO_TCP(q)              (Q_TO_CONN(q)->conn_tcp)
222 #define Q_TO_UDP(q)              (Q_TO_CONN(q)->conn_udp)
223 #define Q_TO_ICMP(q)             (Q_TO_CONN(q)->conn_icmp)
224 #define Q_TO_RTS(q)              (Q_TO_CONN(q)->conn_rts)
225 #define Q_TO_DCCP(q)             (Q_TO_CONN(q)->conn_dccp)
226 #endif /* !codereview */

228 #define CONNP_TO_WQ(connp)        ((connp)->conn_wq)
229 #define CONNP_TO_RQ(connp)        ((connp)->conn_rq)

231 #define GRAB_CONN_LOCK(q)         { \
232     if (q != NULL && CONN_Q(q)) \
233         mutex_enter(&(Q_TO_CONN(q))->conn_lock); \
234 }

236 #define RELEASE_CONN_LOCK(q)      { \
237     if (q != NULL && CONN_Q(q)) \
238         mutex_exit(&(Q_TO_CONN(q))->conn_lock); \
239 }

241 /*
242 * Ref counter macros for ioctls. This provides a guard for TCP to stop
243 * tcp_close from removing the rq/wq whilst an ioctl is still in flight on the
244 * stream. The ioctl could have been queued on e.g. an ipsq. tcp_close will wait
245 * until the ioctlref count is zero before proceeding.
246 * Ideally conn_oper_pending_ill would be used for this purpose. However, in the
247 * case where an ioctl is aborted or interrupted, it can be cleared prematurely.
248 * There are also some race possibilities between ip and the stream head which
249 * can also end up with conn_oper_pending_ill being cleared prematurely. So, to
250 * avoid these situations, we use a dedicated ref counter for ioctls which is
251 * used in addition to and in parallel with the normal conn_ref count.
252 */
253 #define CONN_INC_IOCTLREF_LOCKED(connp) { \
254     ASSERT(MUTEX_HELD(&(connp)->conn_lock)); \
255     DTRACE_PROBE1(conn_inc_ioctlref, conn_t *, (connp)); \
256     (connp)->conn_ioctlref++; \
257     mutex_exit(&(connp)->conn_lock); \
258 }

```

```

260 #define CONN_INC_IOCTLREF(connp) { \
261     mutex_enter(&(connp)->conn_lock); \
262     CONN_INC_IOCTLREF_LOCKED(connp); \
263 }

265 #define CONN_DEC_IOCTLREF(connp) { \
266     mutex_enter(&(connp)->conn_lock); \
267     DTRACE_PROBE1(conn_dec_ioctlref, conn_t *, (connp)); \
268     /* Make sure conn_ioctlref will not underflow. */ \
269     ASSERT((connp)->conn_ioctlref != 0); \
270     if (--(connp)->conn_ioctlref == 0) && \
271         ((connp)->conn_state_flags & CONN_CLOSING)) { \
272         cv_broadcast(&(connp)->conn_cv); \
273     } \
274     mutex_exit(&(connp)->conn_lock); \
275 }

278 /*
279  * Complete the pending operation. Usually an ioctl. Can also
280  * be a bind or option management request that got enqueued
281  * in an ipsq_t. Called on completion of the operation.
282  */
283 #define CONN_OPER_PENDING_DONE(connp) { \
284     mutex_enter(&(connp)->conn_lock); \
285     (connp)->conn_oper_pending_ill = NULL; \
286     cv_broadcast(&(connp)->conn_refcv); \
287     mutex_exit(&(connp)->conn_lock); \
288     CONN_DEC_REF(connp); \
289 }

291 /*
292  * Values for squeue switch:
293  */
294 #define IP_SQUEUE_ENTER_NODRAIN 1
295 #define IP_SQUEUE_ENTER 2
296 #define IP_SQUEUE_FILL 3

298 extern int ip_squeue_flag;

300 /* IP Fragmentation Reassembly Header */
301 typedef struct ipf_s {
302     struct ipf_s *ipf_hash_next;
303     struct ipf_s **ipf_ptphn; /* Pointer to previous hash next. */
304     uint32_t ipf_ident; /* Ident to match. */
305     uint8_t ipf_protocol; /* Protocol to match. */
306     uchar_t ipf_last_frag_seen : 1; /* Last fragment seen ? */
307     time_t ipf_timestamp; /* Reassembly start time. */
308     mblk_t *ipf_mp; /* mblk we live in. */
309     mblk_t *ipf_tail_mp; /* Frag queue tail pointer. */
310     int ipf_hole_cnt; /* Number of holes (hard-case). */
311     int ipf_end; /* Tail end offset (0 -> hard-case). */
312     uint_t ipf_gen; /* Frag queue generation */
313     size_t ipf_count; /* Count of bytes used by frag */
314     uint_t ipf_nf_hdr_len; /* Length of nonfragmented header */
315     in6_addr_t ipf_v6src; /* IPv6 source address */
316     in6_addr_t ipf_v6dst; /* IPv6 dest address */
317     uint_t ipf_prev_nexthdr_offset; /* Offset for nexthdr value */
318     uint8_t ipf_ecn; /* ECN info for the fragments */
319     uint8_t ipf_num_dups; /* Number of times dup frags rcvcd */
320     uint16_t ipf_checksum_flags; /* Hardware checksum flags */
321     uint32_t ipf_checksum; /* Partial checksum of fragment data */
322 } ipf_t;

324 /*
325  * IPv4 Fragments

```

```

326 */
327 #define IS_V4_FRAGMENT(ipha_fragment_offset_and_flags) \
328     (((ntohs(ipha_fragment_offset_and_flags) & IPH_OFFSET) != 0) || \
329     ((ntohs(ipha_fragment_offset_and_flags) & IPH_MF) != 0))

331 #define ipf_src V4_PART_OF_V6(ipf_v6src)
332 #define ipf_dst V4_PART_OF_V6(ipf_v6dst)

334 #endif /* _KERNEL */

336 /* ICMP types */
337 #define ICMP_ECHO_REPLY 0
338 #define ICMP_DEST_UNREACHABLE 3
339 #define ICMP_SOURCE_QUENCH 4
340 #define ICMP_REDIRECT 5
341 #define ICMP_ECHO_REQUEST 8
342 #define ICMP_ROUTER_ADVERTISEMENT 9
343 #define ICMP_ROUTER_SOLICITATION 10
344 #define ICMP_TIME_EXCEEDED 11
345 #define ICMP_PARAM_PROBLEM 12
346 #define ICMP_TIME_STAMP_REQUEST 13
347 #define ICMP_TIME_STAMP_REPLY 14
348 #define ICMP_INFO_REQUEST 15
349 #define ICMP_INFO_REPLY 16
350 #define ICMP_ADDRESS_MASK_REQUEST 17
351 #define ICMP_ADDRESS_MASK_REPLY 18

353 /* Evaluates to true if the ICMP type is an ICMP error */
354 #define ICMP_IS_ERROR(type) ( \
355     (type) == ICMP_DEST_UNREACHABLE || \
356     (type) == ICMP_SOURCE_QUENCH || \
357     (type) == ICMP_TIME_EXCEEDED || \
358     (type) == ICMP_PARAM_PROBLEM)

360 /* ICMP_TIME_EXCEEDED codes */
361 #define ICMP_TTL_EXCEEDED 0
362 #define ICMP_REASSEMBLY_TIME_EXCEEDED 1

364 /* ICMP_DEST_UNREACHABLE codes */
365 #define ICMP_NET_UNREACHABLE 0
366 #define ICMP_HOST_UNREACHABLE 1
367 #define ICMP_PROTOCOL_UNREACHABLE 2
368 #define ICMP_PORT_UNREACHABLE 3
369 #define ICMP_FRAGMENTATION_NEEDED 4
370 #define ICMP_SOURCE_ROUTE_FAILED 5
371 #define ICMP_DEST_NET_UNKNOWN 6
372 #define ICMP_DEST_HOST_UNKNOWN 7
373 #define ICMP_SRC_HOST_ISOLATED 8
374 #define ICMP_DEST_NET_UNREACH_ADMIN 9
375 #define ICMP_DEST_HOST_UNREACH_ADMIN 10
376 #define ICMP_DEST_NET_UNREACH_TOS 11
377 #define ICMP_DEST_HOST_UNREACH_TOS 12

379 /* ICMP Header Structure */
380 typedef struct icmph_s {
381     uint8_t icmph_type;
382     uint8_t icmph_code;
383     uint16_t icmph_checksum;
384     union {
385         struct { /* ECHO request/response structure */
386             uint16_t u_echo_ident;
387             uint16_t u_echo_seqnum;
388         } u_echo;
389         struct { /* Destination unreachable structure */
390             uint16_t u_du_zero;
391             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.

```

```

524 */
525 #define IRE_ONLINK          (IRE_IF_ALL|IRE_LOCAL|IRE_LOOPBACK| \
526                          IRE_BROADCAST|IRE_MULTICAST|IRE_NOROUTE)

528 /* Arguments to ire_flush_cache() */
529 #define IRE_FLUSH_DELETE  0
530 #define IRE_FLUSH_ADD     1
531 #define IRE_FLUSH_GWCHANGE 2

533 /*
534 * Flags to ire_route_recursive
535 */
536 #define IRR_NONE          0
537 #define IRR_ALLOCATE     1      /* OK to allocate IRE_IF_CLONE */
538 #define IRR_INCOMPLETE   2      /* OK to return incomplete chain */

540 /*
541 * Open/close synchronization flags.
542 * These are kept in a separate field in the conn and the synchronization
543 * depends on the atomic 32 bit access to that field.
544 */
545 #define CONN_CLOSING      0x01    /* ip_close waiting for ip_wsrv */
546 #define CONN_CONDEMNED   0x02    /* conn is closing, no more refs */
547 #define CONN_INCIPIENT   0x04    /* conn not yet visible, no refs */
548 #define CONN_QUIESCED    0x08    /* conn is now quiescent */
549 #define CONN_UPDATE_ILM  0x10    /* conn_update_ilm in progress */

551 /*
552 * Flags for dce_flags field. Specifies which information has been set.
553 * dce_ident is always present, but the other ones are identified by the flags.
554 */
555 #define DCEF_DEFAULT     0x0001  /* Default DCE - no pmtu or uinfo */
556 #define DCEF_PMTU        0x0002  /* Different than interface MTU */
557 #define DCEF_UINFO       0x0004  /* dce_uinfo set */
558 #define DCEF_TOO_SMALL_PMTU 0x0008 /* Smaller than IPv4/IPv6 MIN */

560 #ifndef _KERNEL
561 /*
562 * Extra structures need for per-src-addr filtering (IGMPv3/MLDv2)
563 */
564 #define MAX_FILTER_SIZE 64

566 typedef struct slist_s {
567     int          sl_numsrc;
568     in6_addr_t   sl_addr[MAX_FILTER_SIZE];
569 } slist_t;

571 /*
572 * Following struct is used to maintain retransmission state for
573 * a multicast group. One rtx_state_t struct is an in-line field
574 * of the ilm_t struct; the slist_ts in the rtx_state_t struct are
575 * alloc'd as needed.
576 */
577 typedef struct rtx_state_s {
578     uint_t       rtx_timer;      /* retrans timer */
579     int          rtx_cnt;        /* retrans count */
580     int          rtx_fmde_cnt;   /* retrans count for fmde change */
581     slist_t      *rtx_allow;
582     slist_t      *rtx_block;
583 } rtx_state_t;

585 /*
586 * Used to construct list of multicast address records that will be
587 * sent in a single listener report.
588 */
589 typedef struct mrec_s {

```

```

590     struct mrec_s *mrec_next;
591     uint8_t       mrec_type;
592     uint8_t       mrec_auxlen;   /* currently unused */
593     in6_addr_t    mrec_group;
594     slist_t       mrec_srcs;
595 } mrec_t;

597 /* Group membership list per upper conn */

599 /*
600 * We record the multicast information from the socket option in
601 * ilg_ifaddr/ilg_ifindex. This allows rejoining the group in the case when
602 * the ifaddr (or ifindex) disappears and later reappears, potentially on
603 * a different ill. The IPv6 multicast socket options and ioctls all specify
604 * the interface using an ifindex. For IPv4 some socket options/ioctls use
605 * the interface address and others use the index. We record here the method
606 * that was actually used (and leave the other of ilg_ifaddr or ilg_ifindex)
607 * at zero so that we can rejoin the way the application intended.
608 */
609 * We track the ill on which we will or already have joined an ilm using
610 * ilg_ill. When we have succeeded joining the ilm and have a rehold on it
611 * then we set ilg_ilm. Thus intentionally there is a window where ilg_ill is
612 * set and ilg_ilm is not set. This allows clearing ilg_ill as a signal that
613 * the ill is being unplumbed and the ilm should be discarded.
614 */
615 * ilg records the state of multicast memberships of a socket end point.
616 * ilm records the state of multicast memberships with the driver and is
617 * maintained per interface.
618 */
619 * The ilg state is protected by conn_ilg_lock.
620 * The ilg will not be freed until ilg_refcnt drops to zero.
621 */
622 typedef struct ilg_s {
623     struct ilg_s *ilg_next;
624     struct ilg_s **ilg_ptpn;
625     struct conn_s *ilg_connp;      /* Back pointer to get lock */
626     in6_addr_t    ilg_v6group;
627     ipaddr_t      ilg_ifaddr;     /* For some IPv4 cases */
628     uint_t        ilg_ifindex;    /* IPv6 and some other IPv4 cases */
629     struct ill_s  *ilg_ill;       /* Where ilm is joined. No rehold */
630     struct ilm_s  *ilg_ilm;      /* With ilm_rehold */
631     uint_t        ilg_refcnt;
632     mcast_record_t ilg_fmde;     /* MODE_IS_INCLUDE/MODE_IS_EXCLUDE */
633     slist_t       *ilg_filter;
634     boolean_t     ilg_condemned; /* Conceptually deleted */
635 } ilg_t;

637 /*
638 * Multicast address list entry for ill.
639 * ilm_ill is used by IPv4 and IPv6
640 */
641 * The ilm state (and other multicast state on the ill) is protected by
642 * ill_mcast_lock. Operations that change state on both an ilg and ilm
643 * in addition use ill_mcast_serializer to ensure that we can't have
644 * interleaving between e.g., add and delete operations for the same conn_t,
645 * group, and ill. The ill_mcast_serializer is also used to ensure that
646 * multicast group joins do not occur on an interface that is in the process
647 * of joining an IPMP group.
648 */
649 * The comment below (and for other netstack_t references) refers
650 * to the fact that we only do netstack_hold in particular cases,
651 * such as the references from open endpoints (ill_t and conn_t's
652 * pointers). Internally within IP we rely on IP's ability to cleanup e.g.
653 * ire_t's when an ill goes away.
654 */
655 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 adrs 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    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.

```



```

788 */
789 #define CONN_INBOUND_POLICY_PRESENT(connp, ipss) \
790 ((connp)->conn_in_enforce_policy || \
791 (!((connp)->conn_policy_cached) && \
792 (ipss)->ipsec_inbound_v4_policy_present))
793
794 #define CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss) \
795 ((connp)->conn_in_enforce_policy || \
796 (!((connp)->conn_policy_cached) && \
797 (ipss)->ipsec_inbound_v6_policy_present))
798
799 #define CONN_OUTBOUND_POLICY_PRESENT(connp, ipss) \
800 ((connp)->conn_out_enforce_policy || \
801 (!((connp)->conn_policy_cached) && \
802 (ipss)->ipsec_outbound_v4_policy_present))
803
804 #define CONN_OUTBOUND_POLICY_PRESENT_V6(connp, ipss) \
805 ((connp)->conn_out_enforce_policy || \
806 (!((connp)->conn_policy_cached) && \
807 (ipss)->ipsec_outbound_v6_policy_present))
808
809 /*
810 * Information cached in IRE for upper layer protocol (ULP).
811 */
812 typedef struct iulp_s {
813     boolean_t    iulp_set;        /* Is any metric set? */
814     uint32_t     iulp_ssthresh;   /* Slow start threshold (TCP). */
815     clock_t      iulp_rtt;        /* Guestimate in millisecs. */
816     clock_t      iulp_rtt_sd;     /* Cached value of RTT variance. */
817     uint32_t     iulp_spine;      /* Send pipe size. */
818     uint32_t     iulp_rpipe;      /* Receive pipe size. */
819     uint32_t     iulp_rtomax;     /* Max round trip timeout. */
820     uint32_t     iulp_sack;       /* Use SACK option (TCP)? */
821     uint32_t     iulp_mtu;        /* Setable with routing sockets */
822
823     uint32_t
824         iulp_tstamp_ok : 1,      /* Use timestamp option (TCP)? */
825         iulp_wsacle_ok : 1,      /* Use window scale option (TCP)? */
826         iulp_ecn_ok : 1,         /* Enable ECN (for TCP)? */
827         iulp_pmtud_ok : 1,       /* Enable PMTUD? */
828
829         /* These three are passed out by ip_set_destination */
830         iulp_localnet : 1,       /* IRE_ONLINK */
831         iulp_loopback : 1,       /* IRE_LOOPBACK */
832         iulp_local : 1,          /* IRE_LOCAL */
833
834         iulp_not_used : 25;
835 } iulp_t;
836
837 /*
838 * The conn drain list structure (idl_t), protected by idl_lock. Each conn_t
839 * inserted in the list points back at this idl_t using conn_idl, and is
840 * chained by conn_drain_next and conn_drain_prev, which are also protected by
841 * idl_lock. When flow control is relieved, either ip_wsrvt() (STREAMS) or
842 * ill_flow_enable() (non-STREAMS) will call conn_drain().
843 *
844 * The conn drain list, idl_t, itself is part of tx cookie list structure.
845 * A tx cookie list points to a blocked Tx ring and contains the list of
846 * all conn's that are blocked due to the flow-controlled Tx ring (via
847 * the idl drain list). Note that a link can have multiple Tx rings. The
848 * drain list will store the conn's blocked due to Tx ring being flow
849 * controlled.
850 */
851
852 typedef uintptr_t ip_mac_tx_cookie_t;
853 typedef struct idl_s idl_t;

```

```

854 typedef struct idl_tx_list_s idl_tx_list_t;
855
856 struct idl_tx_list_s {
857     ip_mac_tx_cookie_t    txl_cookie;
858     kmutex_t              txl_lock; /* Lock for this list */
859     idl_t                  *txl_drain_list;
860     int                    txl_drain_index;
861 };
862
863 struct idl_s {
864     conn_t                *idl_conn; /* Head of drain list */
865     kmutex_t              idl_lock; /* Lock for this list */
866     idl_tx_list_t        *idl_itl;
867 };
868
869 /*
870 * Interface route structure which holds the necessary information to recreate
871 * routes that are tied to an interface i.e. have ire_ill set.
872 *
873 * These routes which were initially created via a routing socket or via the
874 * SIOCADDRT ioctl may be gateway routes (RTF_GATEWAY being set) or may be
875 * traditional interface routes. When an ill comes back up after being
876 * down, this information will be used to recreate the routes. These
877 * are part of an mblk_t chain that hangs off of the ILL (ill_saved_ire_mp).
878 */
879 typedef struct ifrt_s {
880     ushort_t            ifrt_type; /* Type of IRE */
881     in6_addr_t          ifrt_v6addr; /* Address IRE represents. */
882     in6_addr_t          ifrt_v6gateway_addr; /* Gateway if IRE_OFFLINK */
883     in6_addr_t          ifrt_v6setsrc_addr; /* Src addr if RTF_SETSRC */
884     in6_addr_t          ifrt_v6mask; /* Mask for matching IRE. */
885     uint32_t            ifrt_flags; /* flags related to route */
886     iulp_t              ifrt_metrics; /* Routing socket metrics */
887     zoneid_t            ifrt_zoneid; /* zoneid for route */
888 } ifrt_t;
889
890 #define ifrt_addr V4_PART_OF_V6(ifrt_v6addr)
891 #define ifrt_gateway_addr V4_PART_OF_V6(ifrt_v6gateway_addr)
892 #define ifrt_mask V4_PART_OF_V6(ifrt_v6mask)
893 #define ifrt_setsrc_addr V4_PART_OF_V6(ifrt_v6setsrc_addr)
894
895 /* Number of IP addresses that can be hosted on a physical interface */
896 #define MAX_ADDRS_PER_IF 8192
897 /*
898 * Number of Source addresses to be considered for source address
899 * selection. Used by ipif_select_source_v4/v6.
900 */
901 #define MAX_IPIF_SELECT_SOURCE 50
902
903 #ifdef IP_DEBUG
904 /*
905 * Trace reholds and refreles for debugging.
906 */
907 #define TR_STACK_DEPTH 14
908 typedef struct tr_buf_s {
909     int tr_depth;
910     clock_t tr_time;
911     pc_t tr_stack[TR_STACK_DEPTH];
912 } tr_buf_t;
913
914 typedef struct th_trace_s {
915     int th_refcnt;
916     uint_t th_trace_lastref;
917     kthread_t *th_id;
918 #define TR_BUF_MAX 38
919     tr_buf_t th_trbuf[TR_BUF_MAX];

```

```

920 } th_trace_t;

922 typedef struct th_hash_s {
923     list_node_t    thh_link;
924     mod_hash_t     *thh_hash;
925     ip_stack_t     *thh_ipst;
926 } th_hash_t;
927 #endif

929 /* The following are ipif_state_flags */
930 #define IPIF_CONDEMNED    0x1    /* The ipif is being removed */
931 #define IPIF_CHANGING    0x2    /* A critical ipif field is changing */
932 #define IPIF_SET_LINKLOCAL 0x10 /* transient flag during bringup */

934 /* IP interface structure, one per local address */
935 typedef struct ipif_s {
936     struct ipif_s *ipif_next;
937     struct ill_s *ipif_ill;
938     int ipif_id;
939     in6_addr_t ipif_v6lcl_addr;
940     in6_addr_t ipif_v6subnet;
941     in6_addr_t ipif_v6net_mask;
942     in6_addr_t ipif_v6brd_addr;
943     in6_addr_t ipif_v6pp_dst_addr;
944     uint64_t ipif_flags;
945     uint_t ipif_ire_type;

947     /*
948      * The packet count in the ipif contain the sum of the
949      * packet counts in dead IRE_LOCAL/LOOPBACK for this ipif.
950      */
951     uint_t ipif_ib_pkt_count;

953     /* Exclusive bit fields, protected by ipsq_t */
954     unsigned int
955         ipif_was_up : 1,
956         ipif_addr_ready : 1,
957         ipif_was_dup : 1,
958         ipif_added_nce : 1;

960     ipif_pad_to_31 : 28;

962     ilm_t *ipif_allhosts_ilm;
963     ilm_t *ipif_solmulti_ilm;

965     uint_t ipif_seqid;
966     uint_t ipif_state_flags;
967     uint_t ipif_refcnt;

969     zoneid_t ipif_zoneid;
970     timeout_id_t ipif_recovery_id;
971     boolean_t ipif_trace_disable;

973     /* For an IPMP interface, ipif_bound_ill tracks the ill whose hardware
974      * information this ipif is associated with via ARP/NDP. We can use
975      * an ill pointer (rather than an index) because only ills that are
976      * part of a group will be pointed to, and an ill cannot disappear
977      * while it's in a group.
978      */
979     struct ill_s *ipif_bound_ill;
980     struct ipif_s *ipif_bound_next;
981     boolean_t ipif_bound;

983     struct ire_s *ipif_ire_local;
984     struct ire_s *ipif_ire_if;
985 } ipif_t;

```

```

987 /*
988 * The following table lists the protection levels of the various members
989 * of the ipif_t. The following notation is used.
990 *
991 * Write once - Written to only once at the time of bringing up
992 * the interface and can be safely read after the bringup without any lock.
993 *
994 * ipsq - Need to execute in the ipsq to perform the indicated access.
995 *
996 * ill_lock - Need to hold this mutex to perform the indicated access.
997 *
998 * ill_g_lock - Need to hold this rw lock as reader/writer for read access or
999 * write access respectively.
1000 *
1001 * down ill - Written to only when the ill is down (i.e all ipifs are down)
1002 * up ill - Read only when the ill is up (i.e. at least 1 ipif is up)
1003 *
1004 *
1005 * Table of ipif_t members and their protection
1006 *
1007 * ipif_next                ipsq + ill_lock +      ipsq OR ill_lock OR
1008 *                          ill_g_lock          ill_g_lock
1009 * ipif_ill                 ipsq + down ipif    write once
1010 * ipif_id                  ipsq + down ipif    write once
1011 * ipif_v6lcl_addr         ipsq + down ipif    up ipif
1012 * ipif_v6subnet           ipsq + down ipif    up ipif
1013 * ipif_v6net_mask         ipsq + down ipif    up ipif
1014 *
1015 * ipif_v6brd_addr
1016 * ipif_v6pp_dst_addr
1017 * ipif_flags              ill_lock            ill_lock
1018 * ipif_ire_type           ipsq + down ill     up ill
1019 *
1020 * ipif_ib_pkt_count       Approx
1021 *
1022 * bit fields              ill_lock            ill_lock
1023 *
1024 * ipif_allhosts_ilm       ipsq                ipsq
1025 * ipif_solmulti_ilm       ipsq                ipsq
1026 *
1027 * ipif_seqid              ipsq                Write once
1028 *
1029 * ipif_state_flags        ill_lock            ill_lock
1030 * ipif_refcnt             ill_lock            ill_lock
1031 * ipif_bound_ill         ipsq + ipmp_lock     ipsq OR ipmp_lock
1032 * ipif_bound_next        ipsq                ipsq
1033 * ipif_bound              ipsq                ipsq
1034 * ipif_ire_local          ipsq + ips_ill_g_lock ipsq OR ips_ill_g_lock
1035 * ipif_ire_if            ipsq + ips_ill_g_lock ipsq OR ips_ill_g_lock
1036 */

1038 /*
1039 * Return values from ip_laddr_verify_{v4,v6}
1040 */
1041 typedef enum { IPVL_UNICAST_UP, IPVL_UNICAST_DOWN, IPVL_MCAST, IPVL_BCAST,
1042               IPVL_BAD } ip_laddr_t;

1045 #define IP_TR_HASH(tid) (((uintptr_t)tid) >> 6) & (IP_TR_HASH_MAX - 1)

1047 #ifdef DEBUG
1048 #define IPIF_TRACE_REF(ipif) ipif_trace_ref(ipif)
1049 #define ILL_TRACE_REF(ill) ill_trace_ref(ill)
1050 #define IPIF_UNTRACE_REF(ipif) ipif_untrace_ref(ipif)
1051 #define ILL_UNTRACE_REF(ill) ill_untrace_ref(ill)

```

```

1052 #else
1053 #define IPIF_TRACE_REF(ipif)
1054 #define ILL_TRACE_REF(ill)
1055 #define IPIF_UNTRACE_REF(ipif)
1056 #define ILL_UNTRACE_REF(ill)
1057 #endif

1059 /* IPv4 compatibility macros */
1060 #define ipif_lcl_addr      V4_PART_OF_V6(ipif_v6lcl_addr)
1061 #define ipif_subnet       V4_PART_OF_V6(ipif_v6subnet)
1062 #define ipif_net_mask     V4_PART_OF_V6(ipif_v6net_mask)
1063 #define ipif_brd_addr     V4_PART_OF_V6(ipif_v6brd_addr)
1064 #define ipif_pp_dst_addr  V4_PART_OF_V6(ipif_v6pp_dst_addr)

1066 /* Macros for easy backreferences to the ill. */
1067 #define ipif_isv6         ipif_ill->ill_isv6

1069 #define SIOCCLIFADDR_NDX 112 /* ndx of SIOCCLIFADDR in the ndx ioctl table */

1071 /*
1072 * mode value for ip_ioctl_finish for finishing an ioctl
1073 */
1074 #define CONN_CLOSE      1 /* No mi_copy */
1075 #define COPYOUT         2 /* do an mi_copyout if needed */
1076 #define NO_COPYOUT      3 /* do an mi_copy_done */
1077 #define IPI2MODE(ipi)  ((ipi)->ipi_flags & IPI_GET_CMD ? COPYOUT : NO_COPYOUT)

1079 /*
1080 * The IP-MT design revolves around the serialization objects ipsq_t (IPSQ)
1081 * and ipxop_t (exclusive operation or "xop"). Becoming "writer" on an IPSQ
1082 * ensures that no other threads can become "writer" on any IPSQs sharing that
1083 * IPSQ's xop until the writer thread is done.
1084 *
1085 * Each phyint points to one IPSQ that remains fixed over the phyint's life.
1086 * Each IPSQ points to one xop that can change over the IPSQ's life. If a
1087 * phyint is *not* in an IPMP group, then its IPSQ will refer to the IPSQ's
1088 * "own" xop (ipsq_ownxop). If a phyint *is* part of an IPMP group, then its
1089 * IPSQ will refer to the "group" xop, which is shorthand for the xop of the
1090 * IPSQ of the IPMP meta-interface's phyint. Thus, all phyints that are part
1091 * of the same IPMP group will have their IPSQ's point to the group xop, and
1092 * thus becoming "writer" on any phyint in the group will prevent any other
1093 * writer on any other phyint in the group. All IPSQs sharing the same xop
1094 * are chained together through ipsq_next (in the degenerate common case,
1095 * ipsq_next simply refers to itself). Note that the group xop is guaranteed
1096 * to exist at least as long as there are members in the group, since the IPMP
1097 * meta-interface can only be destroyed if the group is empty.
1098 *
1099 * Incoming exclusive operation requests are enqueued on the IPSQ they arrived
1100 * on rather than the xop. This makes switching xop's (as would happen when a
1101 * phyint leaves an IPMP group) simple, because after the phyint leaves the
1102 * group, any operations enqueued on its IPSQ can be safely processed with
1103 * respect to its new xop, and any operations enqueued on the IPSQs of its
1104 * former group can be processed with respect to their existing group xop.
1105 * Even so, switching xops is a subtle dance; see ipsq_dq() for details.
1106 *
1107 * An IPSQ's "own" xop is embedded within the IPSQ itself since they have a
1108 * identical lifetimes, and because doing so simplifies pointer management.
1109 * While each phyint and IPSQ point to each other, it is not possible to free
1110 * the IPSQ when the phyint is freed, since we may still *inside* the IPSQ
1111 * when the phyint is being freed. Thus, ipsq_phyint is set to NULL when the
1112 * phyint is freed, and the IPSQ free is later done in ipsq_exit().
1113 *
1114 * ipsq_t synchronization:      read          write
1115 *
1116 *     ipsq_xopq_mthead         ipx_lock       ipx_lock
1117 *     ipsq_xopq_mptail         ipx_lock       ipx_lock

```

```

1118 *     ipsq_xop_switch_mp       ipsq_lock     ipsq_lock
1119 *     ipsq_phyint              write once   write once
1120 *     ipsq_next                 RW_READER ill_g_lock RW_WRITER ill_g_lock
1121 *     ipsq_xop                  ipsq_lock or ipsq ipsq_lock + ipsq
1122 *     ipsq_swxop                 ipsq         ipsq
1123 *     ipsq_ownxop               see ipxop_t   see ipxop_t
1124 *     ipsq_ipst                 write once   write once
1125 *
1126 * ipxop_t synchronization:     read          write
1127 *
1128 *     ipx_writer                 ipx_lock     ipx_lock
1129 *     ipx_xop_queued             ipx_lock     ipx_lock
1130 *     ipx_mthead                 ipx_lock     ipx_lock
1131 *     ipx_mptail                 ipx_lock     ipx_lock
1132 *     ipx_ipsq                   write once   write once
1133 *     ips_ipsq_queued            ipx_lock     ipx_lock
1134 *     ipx_waitfor                ipsq or ipx_lock ipsq + ipx_lock
1135 *     ipx_reentry_cnt            ipsq or ipx_lock ipsq + ipx_lock
1136 *     ipx_current_done           ipsq         ipsq
1137 *     ipx_current_ioctl          ipsq         ipsq
1138 *     ipx_current_ipif           ipsq or ipx_lock ipsq + ipx_lock
1139 *     ipx_pending_ipif           ipsq or ipx_lock ipsq + ipx_lock
1140 *     ipx_pending_mp             ipsq or ipx_lock ipsq + ipx_lock
1141 *     ipx_forced                 ipsq         ipsq
1142 *     ipx_depth                  ipsq         ipsq
1143 *     ipx_stack                  ipsq         ipsq
1144 */
1145 typedef struct ipxop_s {
1146     kmutex_t      ipx_lock; /* see above */
1147     kthread_t     *ipx_writer; /* current owner */
1148     mblk_t        *ipx_mthead; /* messages tied to this op */
1149     mblk_t        *ipx_mptail;
1150     struct ipsq_s *ipx_ipsq; /* associated ipsq */
1151     boolean_t     ipx_ipsq_queued; /* ipsq using xop has queued op */
1152     int           ipx_waitfor; /* waiting; values encoded below */
1153     int           ipx_reentry_cnt;
1154     boolean_t     ipx_current_done; /* is the current operation done? */
1155     int           ipx_current_ioctl; /* current ioctl, or 0 if no ioctl */
1156     ipif_t        *ipx_current_ipif; /* ipif for current op */
1157     ipif_t        *ipx_pending_ipif; /* ipif for ipx_pending_mp */
1158     mblk_t        *ipx_pending_mp; /* current ioctl mp while waiting */
1159     boolean_t     ipx_forced; /* debugging aid */
1160 #ifdef DEBUG
1161     int           ipx_depth; /* debugging aid */
1162 #define IPX_STACK_DEPTH 15
1163     pc_t         ipx_stack[IPX_STACK_DEPTH]; /* debugging aid */
1164 #endif
1165 } ipxop_t;

1167 typedef struct ipsq_s {
1168     kmutex_t      ipsq_lock; /* see above */
1169     mblk_t        *ipsq_switch_mp; /* op to handle right after switch */
1170     mblk_t        *ipsq_xopq_mthead; /* list of excl ops (mostly ioctls) */
1171     mblk_t        *ipsq_xopq_mptail;
1172     struct phyint *ipsq_phyint; /* associated phyint */
1173     struct ipsq_s *ipsq_next; /* next ipsq sharing ipsq_xop */
1174     struct ipxop_s *ipsq_xop; /* current xop synchronization info */
1175     struct ipxop_s *ipsq_swxop; /* switch xop to on ipsq_exit() */
1176     struct ipxop_s ipsq_ownxop; /* our own xop (may not be in-use) */
1177     ip_stack_t    *ipsq_ipst; /* does not have a netstack_hold */
1178 } ipsq_t;

1180 /*
1181 * ipx_waitfor values:
1182 */
1183 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_fhtable 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 IRE_MARK_CONDEMNED 0x0001 /* Contains some IRE_IS_CONDEMNED */
1276 #define IRE_MARK_DYNAMIC 0x0002 /* Dynamically allocated */
1277 /* Once IPv6 uses radix then IRE_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_rcv_attr_s;
1307 typedef struct ip_rcv_attr_s ip_rcv_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

```

```

1316 * by reference between ip_input and ill_inputfn.
1317 */
1318 typedef struct {
1319     ire_t         *rtc_ire;
1320     ipaddr_t     rtc_ipaddr;
1321     in6_addr_t   rtc_ip6addr;
1322 } rtc_t;

1324 /*
1325 * Note: Temporarily use 64 bits, and will probably go back to 32 bits after
1326 * more cleanup work is done.
1327 */
1328 typedef uint64_t iaflags_t;

1330 /* The ill input function pointer type */
1331 typedef void (*pfillinput_t)(mblk_t *, void *, void *, ip_rcv_attr_t *,
1332     rtc_t *);

1334 /* The ire receive function pointer type */
1335 typedef void (*pfirerecv_t)(ire_t *, mblk_t *, void *, ip_rcv_attr_t *);

1337 /* The ire send and postfrag function pointer types */
1338 typedef int (*pfiresend_t)(ire_t *, mblk_t *, void *,
1339     ip_xmit_attr_t *, uint32_t *);
1340 typedef int (*pfirepostfrag_t)(mblk_t *, nce_t *, iaflags_t, uint_t, uint32_t,
1341     zoneid_t, zoneid_t, uintptr_t *);

1344 #define IP_V4_G_HEAD    0
1345 #define IP_V6_G_HEAD    1

1347 #define MAX_G_HEADS    2

1349 /*
1350 * unpadded ill_if structure
1351 */
1352 struct _ill_if_s_ {
1353     union ill_if_u    *illif_next;
1354     union ill_if_u    *illif_prev;
1355     avl_tree_t        illif_avl_by_ppa;    /* AVL tree sorted on ppa */
1356     vmem_t            *illif_ppa_arena;    /* ppa index space */
1357     uint16_t          illif_mcast_v1;    /* hints for */
1358     uint16_t          illif_mcast_v2;    /* [igmp/mld]_slowltimo */
1359     int               illif_name_len;    /* name length */
1360     char              illif_name[LIFNAMSIZ]; /* name of interface type */
1361 };

1363 /* cache aligned ill_if structure */
1364 typedef union ill_if_u {
1365     struct _ill_if_s_ ill_if_s;
1366     char illif_filler[CACHE_ALIGN(_ill_if_s)];
1367 } ill_if_t;

1369 #define illif_next        ill_if_s.illif_next
1370 #define illif_prev        ill_if_s.illif_prev
1371 #define illif_avl_by_ppa  ill_if_s.illif_avl_by_ppa
1372 #define illif_ppa_arena  ill_if_s.illif_ppa_arena
1373 #define illif_mcast_v1   ill_if_s.illif_mcast_v1
1374 #define illif_mcast_v2   ill_if_s.illif_mcast_v2
1375 #define illif_name        ill_if_s.illif_name
1376 #define illif_name_len    ill_if_s.illif_name_len

1378 typedef struct ill_walk_context_s {
1379     int    ctx_current_list; /* current list being searched */
1380     int    ctx_last_list; /* last list to search */
1381 } ill_walk_context_t;

```

```

1383 /*
1384 * ill_g_heads structure, one for IPV4 and one for IPV6
1385 */
1386 struct _ill_g_head_s_ {
1387     ill_if_t    *ill_g_list_head;
1388     ill_if_t    *ill_g_list_tail;
1389 };

1391 typedef union ill_g_head_u {
1392     struct _ill_g_head_s_ ill_g_head_s;
1393     char    ill_g_head_filler[CACHE_ALIGN(_ill_g_head_s)];
1394 } ill_g_head_t;

1396 #define ill_g_list_head ill_g_head_s.ill_g_list_head
1397 #define ill_g_list_tail ill_g_head_s.ill_g_list_tail

1399 #define IP_V4_ILL_G_LIST(ipst) \
1400     (ipst)->ips_ill_g_heads[IP_V4_G_HEAD].ill_g_list_head
1401 #define IP_V6_ILL_G_LIST(ipst) \
1402     (ipst)->ips_ill_g_heads[IP_V6_G_HEAD].ill_g_list_head
1403 #define IP_VX_ILL_G_LIST(i, ipst) \
1404     (ipst)->ips_ill_g_heads[i].ill_g_list_head

1406 #define ILL_START_WALK_V4(ctx_ptr, ipst) \
1407     ill_first(IP_V4_G_HEAD, IP_V4_G_HEAD, ctx_ptr, ipst)
1408 #define ILL_START_WALK_V6(ctx_ptr, ipst) \
1409     ill_first(IP_V6_G_HEAD, IP_V6_G_HEAD, ctx_ptr, ipst)
1410 #define ILL_START_WALK_ALL(ctx_ptr, ipst) \
1411     ill_first(MAX_G_HEADS, MAX_G_HEADS, ctx_ptr, ipst)

1413 /*
1414 * Capabilities, possible flags for ill_capabilities.
1415 */
1416 #define ILL_CAPAB_LSO            0x04    /* Large Send Offload */
1417 #define ILL_CAPAB_HCKSUM        0x08    /* Hardware checksumming */
1418 #define ILL_CAPAB_ZEROCOPY      0x10    /* Zero-copy */
1419 #define ILL_CAPAB_DLD           0x20    /* DLD capabilities */
1420 #define ILL_CAPAB_DLD_POLL      0x40    /* Polling */
1421 #define ILL_CAPAB_DLD_DIRECT    0x80    /* Direct function call */

1423 /*
1424 * Per-ill Hardware Checksumming capabilities.
1425 */
1426 typedef struct ill_hcksum_capab_s ill_hcksum_capab_t;

1428 /*
1429 * Per-ill Zero-copy capabilities.
1430 */
1431 typedef struct ill_zero copy_capab_s ill_zero copy_capab_t;

1433 /*
1434 * DLD capabilities.
1435 */
1436 typedef struct ill_dld_capab_s ill_dld_capab_t;

1438 /*
1439 * Per-ill polling resource map.
1440 */
1441 typedef struct ill_rx_ring ill_rx_ring_t;

1443 /*
1444 * Per-ill Large Send Offload capabilities.
1445 */
1446 typedef struct ill_lso_capab_s ill_lso_capab_t;

```

```

1448 /* The following are ill_state_flags */
1449 #define ILL_LL_SUBNET_PENDING 0x01 /* Waiting for DL_INFO_ACK from drv */
1450 #define ILL_CONDEMNED 0x02 /* No more new ref's to the ILL */
1451 #define ILL_DL_UNBIND_IN_PROGRESS 0x04 /* UNBIND_REQ is sent */
1452 /*
1453 * ILL_DOWN_IN_PROGRESS is set to ensure the following:
1454 * - no packets are sent to the driver after the DL_UNBIND_REQ is sent,
1455 * - no longstanding references will be acquired on objects that are being
1456 * brought down.
1457 */
1458 #define ILL_DOWN_IN_PROGRESS 0x08

1460 /* Is this an ILL whose source address is used by other ILL's ? */
1461 #define IS_USESRC_ILL(ill) \
1462 ((ill->ill_usersrc_ifindex == 0) && \
1463 ((ill->ill_usersrc_grp_next != NULL))

1465 /* Is this a client/consumer of the usersrc ILL ? */
1466 #define IS_USESRC_CLT_ILL(ill) \
1467 ((ill->ill_usersrc_ifindex != 0) && \
1468 ((ill->ill_usersrc_grp_next != NULL))

1470 /* Is this a virtual network interface (vni) ILL ? */
1471 #define IS_VNI(ill) \
1472 (((ill->ill_phyint->phyint_flags & (PHYI_LOOPBACK|PHYI_VIRTUAL)) == \
1473 PHYI_VIRTUAL)

1475 /* Is this a loopback ILL? */
1476 #define IS_LOOPBACK(ill) \
1477 ((ill->ill_phyint->phyint_flags & PHYI_LOOPBACK)

1479 /* Is this an IPMP meta-interface ILL? */
1480 #define IS_IPMP(ill) \
1481 ((ill->ill_phyint->phyint_flags & PHYI_IPMP)

1483 /* Is this ILL under an IPMP meta-interface? (aka "in a group?") */
1484 #define IS_UNDER_IPMP(ill) \
1485 ((ill->ill_grp != NULL && !IS_IPMP(ill))

1487 /* Is ill1 in the same illgrp as ill2? */
1488 #define IS_IN_SAME_ILLGRP(ill1, ill2) \
1489 ((ill1->ill_grp != NULL && ((ill1->ill_grp == (ill2->ill_grp))

1491 /* Is ill1 on the same LAN as ill2? */
1492 #define IS_ON_SAME_LAN(ill1, ill2) \
1493 ((ill1) == (ill2) || IS_IN_SAME_ILLGRP(ill1, ill2))

1495 #define ILL_OTHER(ill) \
1496 ((ill->ill_isv6 ? (ill->ill_phyint->phyint_illv4 : \
1497 (ill->ill_phyint->phyint_illv6)

1499 /*
1500 * IPMP group ILL state structure -- up to two per IPMP group (V4 and V6).
1501 * Created when the V4 and/or V6 IPMP meta-interface is I_PLINK'd. It is
1502 * guaranteed to persist while there are interfaces of that type in the group.
1503 * In general, most fields are accessed outside of the IPSQ (e.g., in the
1504 * datapath), and thus use locks in addition to the IPSQ for protection.
1505 *
1506 * synchronization: read write
1507 *
1508 * ig_if ipsq or ill_g_lock ipsq and ill_g_lock
1509 * ig_activ ipsq or ipmp_lock ipsq and ipmp_lock
1510 * ig_nactif ipsq or ipmp_lock ipsq and ipmp_lock
1511 * ig_next_ill ipsq or ipmp_lock ipsq and ipmp_lock
1512 * ig_ipmp_ill write once write once
1513 * ig_cast_ill ipsq or ipmp_lock ipsq and ipmp_lock

```

```

1514 * ig_arpent ipsq ipsq
1515 * ig_mtu ipsq ipsq
1516 * ig_mc_mtu ipsq ipsq
1517 */
1518 typedef struct ipmp_illgrp_s {
1519 list_t ig_if; /* list of all interfaces */
1520 list_t ig_activ; /* list of active interfaces */
1521 uint_t ig_nactif; /* number of active interfaces */
1522 struct ill_s *ig_next_ill; /* next active interface to use */
1523 struct ill_s *ig_ipmp_ill; /* backpointer to IPMP meta-interface */
1524 struct ill_s *ig_cast_ill; /* nominated ill for multi/broadcast */
1525 list_t ig_arpent; /* list of ARP entries */
1526 uint_t ig_mtu; /* ig_ipmp_ill->ill_mtu */
1527 uint_t ig_mc_mtu; /* ig_ipmp_ill->ill_mc_mtu */
1528 } ipmp_illgrp_t;

1530 /*
1531 * IPMP group state structure -- one per IPMP group. Created when the
1532 * IPMP meta-interface is plumbed; it is guaranteed to persist while there
1533 * are interfaces in it.
1534 *
1535 * ipmp_grp_t synchronization: read write
1536 *
1537 * gr_name ipmp_lock ipmp_lock
1538 * gr_ifname write once write once
1539 * gr_mactype ipmp_lock ipmp_lock
1540 * gr_phyint write once write once
1541 * gr_nif ipmp_lock ipmp_lock
1542 * gr_nactif ipsq ipsq
1543 * gr_v4 ipmp_lock ipmp_lock
1544 * gr_v6 ipmp_lock ipmp_lock
1545 * gr_nv4 ipmp_lock ipmp_lock
1546 * gr_nv6 ipmp_lock ipmp_lock
1547 * gr_pendv4 ipmp_lock ipmp_lock
1548 * gr_pendv6 ipmp_lock ipmp_lock
1549 * gr_linkdownmp ipsq ipsq
1550 * gr_ksp ipmp_lock ipmp_lock
1551 * gr_kstats0 atomic atomic
1552 */
1553 typedef struct ipmp_grp_s {
1554 char gr_name[LIFGRNAMSIZ]; /* group name */
1555 char gr_ifname[LIFNAMSIZ]; /* interface name */
1556 t_uscalar_t gr_mactype; /* DLPI mactype of group */
1557 phyint_t *gr_phyint; /* IPMP group phyint */
1558 uint_t gr_nif; /* number of interfaces in group */
1559 uint_t gr_nactif; /* number of active interfaces */
1560 ipmp_illgrp_t *gr_v4; /* V4 group information */
1561 ipmp_illgrp_t *gr_v6; /* V6 group information */
1562 uint_t gr_nv4; /* number of ill_s in V4 group */
1563 uint_t gr_nv6; /* number of ill_s in V6 group */
1564 uint_t gr_pendv4; /* number of pending ill_s in V4 group */
1565 uint_t gr_pendv6; /* number of pending ill_s in V6 group */
1566 mblk_t *gr_linkdownmp; /* message used to bring link down */
1567 kstat_t *gr_ksp; /* group kstat pointer */
1568 uint64_t gr_kstats0[IPMP_KSTAT_MAX]; /* baseline group kstats */
1569 } ipmp_grp_t;

1571 /*
1572 * IPMP ARP entry -- one per SIOCS*ARP entry tied to the group. Used to keep
1573 * ARP up-to-date as the active set of interfaces in the group changes.
1574 */
1575 typedef struct ipmp_arpent_s {
1576 ipaddr_t ia_ipaddr; /* IP address for this entry */
1577 boolean_t ia_proxyparp; /* proxy ARP entry? */
1578 boolean_t ia_notified; /* ARP notified about this entry? */
1579 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_uscalar_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_cred; /* 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 ipsqt */
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_llla* 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_llla_mp; /* mblk which holds ill_nd_llla */
1765 uint8_t  *ill_nd_llla; /* Link Layer Address */
1766 uint_t  ill_nd_llla_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_usersrc_ifindex; /* use src addr from this ILL */
1794     struct ill_s  *ill_usersrc_grp_next; /* Next ILL in the usersrc group */
1795     boolean_t  ill_trace_disable; /* True when alloc fails */
1796     ill_zoneid_t  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     ill_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_mrouterr_cnt; /* mrouterr 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

```



```

1844 * to that ipif/ill goes to zero. ILL_DOWN_OK() is a necessary condition
1845 * quiescence checks. See comments above IPIF_DOWN_OK for details
1846 * on why ires and nces are selectively considered for this macro.
1847 */
1848 #define ILL_DOWN_OK(ill)
1849 (ill->ill_ire_cnt == 0 && ill->ill_ncec_cnt == 0 &&
1850  ill->ill_nce_cnt == 0)

1852 /*
1853 * The following table lists the protection levels of the various members
1854 * of the ill_t. Same notation as that used for ipif_t above is used.
1855 *
1856 *           Write           Read
1857 *
1858 * ill_ifptr           ill_g_lock + s           Write once
1859 * ill_rq              ipsq                     Write once
1860 * ill_wq              ipsq                     Write once
1861 *
1862 * ill_error           ipsq                     None
1863 * ill_ipif            ill_g_lock + ipsq        ill_g_lock OR ipsq
1864 * ill_ipif_up_count  ill_lock + ipsq         ill_lock OR ipsq
1865 * ill_max_frag        ill_lock                ill_lock
1866 * ill_current_frag   ill_lock                ill_lock
1867 *
1868 * ill_name            ill_g_lock + ipsq        Write once
1869 * ill_name_length    ill_g_lock + ipsq        Write once
1870 * ill_ndd_name        ipsq                     Write once
1871 * ill_net_type        ipsq                     Write once
1872 * ill_ppa             ill_g_lock + ipsq        Write once
1873 * ill_sap             ipsq + down ill          Write once
1874 * ill_sap_length      ipsq + down ill          Write once
1875 * ill_phys_addr_length ipsq + down ill        Write once
1876 *
1877 * ill_bcast_addr_length ipsq                 ipsq
1878 * ill_mactype         ipsq                 ipsq
1879 * ill_frag_ptr        ipsq                 ipsq
1880 *
1881 * ill_frag_timer_id   ill_lock              ill_lock
1882 * ill_frag_hash_tbl   ipsq                  up ill
1883 * ill_ilm              ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1884 * ill_global_timer    ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1885 * ill_mcast_type       ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1886 * ill_mcast_v1_time    ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1887 * ill_mcast_v2_time    ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1888 * ill_mcast_v1_tset    ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1889 * ill_mcast_v2_tset    ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1890 * ill_mcast_rv         ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1891 * ill_mcast_qi         ill_mcast_lock(WRITER) ill_mcast_lock(READER)
1892 *
1893 * ill_down_mp         ipsq                  ipsq
1894 * ill_dlpi_deferred   ill_lock              ill_lock
1895 * ill_dlpi_pending    ipsq + ill_lock        ipsq or ill_lock or
1896 *                       absence of ipsq writer.
1897 * ill_phys_addr_mp    ipsq + down ill        only when ill is up
1898 * ill_mcast_deferred  ill_lock              ill_lock
1899 * ill_phys_addr       ipsq + down ill        only when ill is up
1900 * ill_dest_addr_mp    ipsq + down ill        only when ill is up
1901 * ill_dest_addr       ipsq + down ill        only when ill is up
1902 *
1903 * ill_state_flags     ill_lock              ill_lock
1904 * exclusive bit flags ipsq_t                ipsq_t
1905 * shared bit flags    ill_lock              ill_lock
1906 *
1907 * ill_muxid           ipsq                  Not atomic
1908 *
1909 * ill_ipf_gen         Not atomic

```

```

1910 * ill_frag_count      atomics                atomics
1911 * ill_type             ipsq + down ill        only when ill is up
1912 * ill_dlpi_multicast_state ill_lock          ill_lock
1913 * ill_dlpi_fastpath_state ill_lock          ill_lock
1914 * ill_dlpi_capab_state ipsq                 ipsq
1915 * ill_max_hops         ipsq                 Not atomic
1916 *
1917 * ill_mtu              ill_lock              None
1918 * ill_mc_mtu          ill_lock              None
1919 *
1920 * ill_user_mtu        ipsq + ill_lock        ill_lock
1921 * ill_reachable_time  ipsq + ill_lock        ill_lock
1922 * ill_reachable_retrans_time ipsq + ill_lock ill_lock
1923 * ill_max_buf         ipsq + ill_lock        ill_lock
1924 *
1925 * Next 2 fields need ill_lock because of the get ioctls. They should not
1926 * report partially updated results without executing in the ipsq.
1927 * ill_token           ipsq + ill_lock        ill_lock
1928 * ill_token_length    ipsq + ill_lock        ill_lock
1929 * ill_dest_token      ipsq + down ill        only when ill is up
1930 * ill_xmit_count      ipsq + down ill        write once
1931 * ill_ip6_mib         ipsq + down ill        only when ill is up
1932 * ill_icmp6_mib      ipsq + down ill        only when ill is up
1933 *
1934 * ill_phyint          ipsq, ill_g_lock, ill_lock Any of them
1935 * ill_flags           ill_lock              ill_lock
1936 * ill_nd_lls_mp       ipsq + down ill        only when ill is up
1937 * ill_nd_lls         ipsq + down ill        only when ill is up
1938 * ill_nd_lls_len      ipsq + down ill        only when ill is up
1939 * ill_phys_addr_pend ipsq + down ill        only when ill is up
1940 * ill_ifname_pending_err ipsq                 ipsq
1941 * ill_avl_byppa       ipsq, ill_g_lock        write once
1942 *
1943 * ill_fastpath_list   ill_lock              ill_lock
1944 * ill_refcnt          ill_lock              ill_lock
1945 * ill_ire_cnt         ill_lock              ill_lock
1946 * ill_cv              ill_lock              ill_lock
1947 * ill_ncec_cnt        ill_lock              ill_lock
1948 * ill_nce_cnt         ill_lock              ill_lock
1949 * ill_ilm_cnt         ill_lock              ill_lock
1950 * ill_src_ipif        ill_g_lock              ill_g_lock
1951 * ill_trace           ill_lock              ill_lock
1952 * ill_usesrc_grp_next ill_g_usesrc_lock        ill_g_usesrc_lock
1953 * ill_dhcpinit        atomics                atomics
1954 * ill_flownotify_mh   write once            write once
1955 * ill_capab_pending_cnt ipsq                 ipsq
1956 * ill_ipallmulti_cnt  ill_lock              ill_lock
1957 * ill_ipallmulti_ilm  ill_lock              ill_lock
1958 * ill_saved_ire_mp    ill_saved_ire_lock    ill_saved_ire_lock
1959 * ill_saved_ire_cnt   ill_saved_ire_lock    ill_saved_ire_lock
1960 * ill_arl             ???                  ???
1961 * ill_ire_multicast   ipsq + quiescent        none
1962 * ill_bound_ipif      ipsq                 ipsq
1963 * ill_actnode         ipsq + ipmp_lock        ipsq OR ipmp_lock
1964 * ill_grpnode         ipsq + ill_g_lock        ipsq OR ill_g_lock
1965 * ill_src_ipif        ill_g_lock              ill_g_lock
1966 * ill_move_ipif       ipsq                 ipsq
1967 * ill_nom_cast        ipsq                 ipsq OR advisory
1968 * ill_refresh_tid     ill_lock              ill_lock
1969 * ill_grp (for IPMP ill) write once            write once
1970 * ill_grp (for underlying ill) ipsq + ill_g_lock ipsq OR ill_g_lock
1971 * ill_grp_pending     ill_mcast_serializer  ill_mcast_serializer
1972 * ill_mrouter_cnt     atomics                atomics
1973 * ill_mphysaddr_list ill_lock              ill_lock
1974 *
1975 * NOTE: It's OK to make heuristic decisions on an underlying interface

```

```

1976 *           by using IS_UNDER_IPMP() or comparing ill_grp's raw pointer value.
1977 */
1979 /*
1980 * For ioctl restart mechanism see ip_reprocess_ioctl()
1981 */
1982 struct ip_ioctl_cmd_s;

1984 typedef int (*ifunc_t)(ipif_t *, struct sockaddr_in *, queue_t *, mblk_t *,
1985     struct ip_ioctl_cmd_s *, void *);

1987 typedef struct ip_ioctl_cmd_s {
1988     int     ipi_cmd;
1989     size_t  ipi_copyin_size;
1990     uint_t  ipi_flags;
1991     uint_t  ipi_cmd_type;
1992     ifunc_t ipi_func;
1993     ifunc_t ipi_func_restart;
1994 } ip_ioctl_cmd_t;

1996 /*
1997 * ipi_cmd_type:
1998 *
1999 * IF_CMD           1      old style ifreq cmd
2000 * LIF_CMD          2      new style lifreq cmd
2001 * ARP_CMD          3      arpreq cmd
2002 * XARP_CMD         4      xarpreq cmd
2003 * MSFILT_CMD       5      multicast source filter cmd
2004 * MISC_CMD         6      misc cmd (not a more specific one above)
2005 */

2007 enum { IF_CMD = 1, LIF_CMD, ARP_CMD, XARP_CMD, MSFILT_CMD, MISC_CMD };

2009 #define IPI_DONTCARE    0      /* For ioctl encoded values that don't matter */

2011 /* Flag values in ipi_flags */
2012 #define IPI_PRIV       0x1    /* Root only command */
2013 #define IPI_MODOK      0x2    /* Permitted on mod instance of IP */
2014 #define IPI_WR         0x4    /* Need to grab writer access */
2015 #define IPI_GET_CMD    0x8    /* branch to mi_copyout on success */
2016 /* unused            0x10    */
2017 #define IPI_NULL_BCONT 0x20    /* ioctl has not data and hence no b_cont */

2019 extern ip_ioctl_cmd_t  ip_ndx_ioctl_table[];
2020 extern ip_ioctl_cmd_t  ip_misc_ioctl_table[];
2021 extern int ip_ndx_ioctl_count;
2022 extern int ip_misc_ioctl_count;

2024 /* Passed down by ARP to IP during I_PLINK/I_PUNLINK */
2025 typedef struct ipmx_s {
2026     char  ipmx_name[LIFNAMSIZ];      /* if name */
2027     uint_t
2028     ipmx_arpdev_stream : 1,          /* This is the arp stream */
2029     ipmx_notused : 31;
2030 } ipmx_t;

2032 /*
2033 * State for detecting if a driver supports certain features.
2034 * Support for DL_ENABMULTI_REQ uses ill_dlpi_multicast_state.
2035 * Support for DLPI M_DATA fastpath uses ill_dlpi_fastpath_state.
2036 */
2037 #define IDS_UNKNOWN    0      /* No DLPI request sent */
2038 #define IDS_INPROGRESS 1      /* DLPI request sent */
2039 #define IDS_OK         2      /* DLPI request completed successfully */
2040 #define IDS_FAILED     3      /* DLPI request failed */

```

```

2042 /* Support for DL_CAPABILITY_REQ uses ill_dlpi_capab_state. */
2043 enum {
2044     IDCS_UNKNOWN,
2045     IDCS_PROBE_SENT,
2046     IDCS_OK,
2047     IDCS_RESET_SENT,
2048     IDCS_RENEG,
2049     IDCS_FAILED
2050 };

2052 /* Extended NDP Management Structure */
2053 typedef struct ipndp_s {
2054     ndgetf_t    ip_ndp_getf;
2055     ndsetf_t    ip_ndp_setf;
2056     caddr_t     ip_ndp_data;
2057     char        *ip_ndp_name;
2058 } ipndp_t;

2060 /* IXA Notification types */
2061 typedef enum {
2062     IXAN_LSO,          /* LSO capability change */
2063     IXAN_PMTU,        /* PMTU change */
2064     IXAN_ZCOPY,       /* ZEROCOPY capability change */
2065 } ixa_notify_type_t;

2067 typedef uint_t ixa_notify_arg_t;

2069 typedef void     (*ixa_notify_t)(void *, ip_xmit_attr_t *ixa, ixa_notify_type_t,
2070     ixa_notify_arg_t);

2072 /*
2073 * Attribute flags that are common to the transmit and receive attributes
2074 */
2075 #define IAF_IS_IPV4          0x80000000 /* ipsec_v4 */
2076 #define IAF_TRUSTED_ICMP    0x40000000 /* ipsec_icmp_loopback */
2077 #define IAF_NO_LOOP_ZONEID_SET 0x20000000 /* Zone that shouldn't have */
2078 /* a copy */
2079 #define IAF_LOOPBACK_COPY   0x10000000 /* For multi and broadcast */

2081 #define IAF_MASK            0xf0000000 /* Flags that are common */

2083 /*
2084 * Transmit side attributes used between the transport protocols and IP as
2085 * well as inside IP. It is also used to cache information in the conn_t i.e.
2086 * replaces conn_ire and the IPsec caching in the conn_t.
2087 */
2088 struct ip_xmit_attr_s {
2089     iaflags_t    ixa_flags;      /* IXAF_*. See below */
2091     uint32_t     ixa_free_flags; /* IXA_FREE*. See below */
2092     uint32_t     ixa_refcnt;     /* Using atomics */

2094     /*
2095      * Always initialized independently of ixa_flags settings.
2096      * Used by ip_xmit so we keep them up front for cache locality.
2097      */
2098     uint32_t     ixa_xmit_hint;  /* For ECMP and GLD TX ring fanout */
2099     uint_t       ixa_pktlen;     /* Always set. For frag and stats */
2100     zoneid_t     ixa_zoneid;    /* Assumed always set */

2102     /* Always set for conn_ip_output(); might be stale */
2103     /*
2104      * Since TCP keeps the conn_t around past the process going away
2105      * we need to use the "notr" (e.g, ire_refhold_notr) for ixa_ire,
2106      * ixa_nce, and ixa_dce.
2107      */

```

```

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_BROADCAST_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_cpuid; /* For getpeerucred */

2193 #ifdef DEBUG
2194     kthread_t *ixa_curthread; /* For serialization assert */
2195 #endif
2196     squeue_t *ixa_sq; /* Set from conn_sq 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 is 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 */

```

```

2240 #define IXAF_SCOPEID_SET      0x00800000    /* ixa_scopeid set */

2242 #define IXAF_MULTIRT_MULTICAST 0x01000000    /* MULTIRT for multicast */
2243 #define IXAF_NO_HW_CKSUM       0x02000000    /* Force software cksum */
2244 #define IXAF_SET_RAW_CKSUM     0x04000000    /* Use ixa_raw_cksum_offset */
2245 #define IXAF_IPSEC_GLOBAL_POLICY 0x08000000   /* Policy came from global */

2247 /* Note the following uses bits 0x10000000 through 0x80000000 */
2248 #define IXAF_IS_IPV4           IAF_IS_IPV4
2249 #define IXAF_TRUSTED_ICMP      IAF_TRUSTED_ICMP
2250 #define IXAF_NO_LOOP_ZONEID_SET IAF_NO_LOOP_ZONEID_SET
2251 #define IXAF_LOOPBACK_COPY     IAF_LOOPBACK_COPY

2253 /* Note: use the upper 32 bits */
2254 #define IXAF_VERIFY_LSO        0x100000000    /* Check LSO capability */
2255 #define IXAF_LSO_CAPAB        0x200000000    /* Capable of LSO */
2256 #define IXAF_VERIFY_ZCOPY     0x400000000    /* Check Zero Copy capability */
2257 #define IXAF_ZCOPY_CAPAB      0x800000000    /* Capable of ZEROCOPY */

2259 /*
2260 * The normal flags for sending packets e.g., icmp errors
2261 */
2262 #define IXAF_BASIC_SIMPLE_V4 \
2263 (IXAF_SET_ULP_CKSUM | IXAF_IS_IPV4 | IXAF_VERIFY_SOURCE)
2264 #define IXAF_BASIC_SIMPLE_V6 \
2265 (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE)

2266 /*
2267 * Normally these fields do not have a hold. But in some cases they do, for
2268 * instance when we've gone through ip_*_attr_to/from_mblk.
2269 * We use ixa_free_flags to indicate that they have a hold and need to be
2270 * released on cleanup.
2271 */
2272 #define IXA_FREE_CRED          0x00000001    /* ixa_cred needs to be rele */
2273 #define IXA_FREE_TSL          0x00000002    /* ixa_tsl needs to be rele */

2275 /*
2276 * Simplistic way to set the ixa_xmit_hint for locally generated traffic
2277 * and forwarded traffic. The shift amount are based on the size of the
2278 * structs to discard the low order bits which don't have much if any variation
2279 * (coloring in kmem_cache_alloc might provide some variation).
2280 *
2281 * Basing the locally generated hint on the address of the conn_t means that
2282 * the packets from the same socket/connection do not get reordered.
2283 * Basing the hint for forwarded traffic on the ill_ring_t means that
2284 * packets from the same NIC+ring are likely to use the same outbound ring
2285 * hence we get low contention on the ring in the transmitting driver.
2286 */
2287 #define CONN_TO_XMIT_HINT(connp)  ((uint32_t)((uintptr_t)connp) >> 11))
2288 #define ILL_RING_TO_XMIT_HINT(ring) ((uint32_t)((uintptr_t)ring) >> 7))

2290 /*
2291 * IP set Destination Flags used by function ip_set_destination,
2292 * ip_attr_connect, and conn_connect.
2293 */
2294 #define IPDF_ALLOW_MCBC        0x1    /* Allow multi/broadcast */
2295 #define IPDF_VERIFY_DST        0x2    /* Verify destination addr */
2296 #define IPDF_SELECT_SRC        0x4    /* Select source address */
2297 #define IPDF_LSO                0x8    /* Try LSO */
2298 #define IPDF_IPSEC              0x10   /* Set IPsec policy */
2299 #define IPDF_ZONE_IS_GLOBAL     0x20   /* From conn_zone_is_global */
2300 #define IPDF_ZCOPY              0x40   /* Try ZEROCOPY */
2301 #define IPDF_UNIQUE_DCE         0x80   /* Get a per-destination DCE */

2303 /*
2304 * Receive side attributes used between the transport protocols and IP as
2305 * well as inside IP.

```

```

2306 */
2307 struct ip_rcv_attr_s {
2308     iaflags_t    ira_flags;    /* See below */

2310     uint32_t     ira_free_flags; /* IRA_FREE*. See below */

2312     /*
2313      * This is a hint for TCP SYN packets.
2314      * Always initialized independently of ira_flags settings
2315      */
2316     queue_t      *ira_sqp;
2317     ill_rx_ring_t *ira_ring;    /* Internal to IP */

2319     /* For ip_accept_tcp when IRAF_TARGET_SQP is set */
2320     queue_t      *ira_target_sqp;
2321     mblk_t        *ira_target_sqp_mp;

2323     /* Always initialized independently of ira_flags settings */
2324     uint32_t     ira_xmit_hint; /* For ECMP and GLD TX ring fanout */
2325     zoneid_t     ira_zoneid;    /* ALL_ZONES unless local delivery */
2326     uint_t       ira_pktlen;    /* Always set. For frag and stats */
2327     uint16_t     ira_ip_hdr_length; /* Points to ULP header */
2328     uint8_t      ira_protocol;  /* Protocol number for ULP cksum */
2329     uint_t       ira_rifindex;  /* Received ifindex */
2330     uint_t       ira_ruifindex; /* Received upper ifindex */
2331     ts_label_t   *ira_tsl;     /* Always set. NULL if not TX */
2332     /*
2333      * ira_rill and ira_ill is set inside IP, but not when conn_rcv is
2334      * called; ULPs should use ira_ruifindex instead.
2335      */
2336     ill_t        *ira_rill;     /* ill where packet came */
2337     ill_t        *ira_ill;     /* ill where IP address hosted */
2338     cred_t       *ira_cred;    /* For getpeerucred */
2339     pid_t        ira_cpuid;    /* For getpeerucred */

2341     /* Used when IRAF_VERIFIED_SRC is set; this source was ok */
2342     ipaddr_t     ira_verified_src;

2344     /*
2345      * The following IPsec fields are only initialized when
2346      * IRAF_IPSEC_SECURE is set. Otherwise they contain garbage.
2347      */
2348     struct ipsec_action_s *ira_ipsec_action; /* how we made it in.. */
2349     struct ipsa_s         *ira_ipsec_ah_sa;  /* SA for AH */
2350     struct ipsa_s         *ira_ipsec_esp_sa; /* SA for ESP */

2352     ipaddr_t     ira_mroute_tunnel; /* IRAF_MROUTE_TUNNEL_SET */

2354     zoneid_t     ira_no_loop_zoneid; /* IRAF_NO_LOOP_ZONEID_SET */

2356     uint32_t     ira_esp_udp_ports; /* IRAF_ESP_UDP_PORTS */

2358     /*
2359      * For IP_RECVSLLA and ip_ndp_conflict/find_solicitation.
2360      * Same size as max for sockaddr_dl
2361      */
2362     #define IRA_L2SRC_SIZE 244
2363     uint8_t      ira_l2src[IRA_L2SRC_SIZE]; /* If IRAF_L2SRC_SET */

2365     /*
2366      * Local handle that we use to do lazy setting of ira_l2src.
2367      * We defer setting l2src until needed but we do before any
2368      * ip_input pullupmsg or copymsg.
2369      */
2370     struct mac_header_info_s *ira_mhip; /* Could be NULL */
2371 };

```

```

2373 /*
2374  * Flags to indicate which receive attributes are set.
2375  */
2376 #define IRAF_SYSTEM_LABELED      0x00000001 /* is_system_labeled() */
2377 #define IRAF_IPV4_OPTIONS        0x00000002 /* Performance */
2378 #define IRAF_MULTICAST           0x00000004 /* Was multicast at L3 */
2379 #define IRAF_BROADCAST           0x00000008 /* Was broadcast at L3 */
2380 #define IRAF_MULTIBROADCAST      (IRAF_MULTICAST|IRAF_BROADCAST)

2382 #define IRAF_LOOPBACK            0x00000010 /* Looped back by IP */
2383 #define IRAF_VERIFY_IP_CKSUM     0x00000020 /* Need to verify IP */
2384 #define IRAF_VERIFY_ULP_CKSUM    0x00000040 /* Need to verify TCP,UDP,etc */
2385 #define IRAF_SCTP_CSUM_ERR       0x00000080 /* sctp pkt has failed cksum */

2387 #define IRAF_IPSEC_SECURE        0x00000100 /* Passed AH and/or ESP */
2388 #define IRAF_DHCP_UNICAST        0x00000200
2389 #define IRAF_IPSEC_DECAPS        0x00000400 /* Was packet decapsulated */
2390 /* from a matching inner packet? */
2391 #define IRAF_TARGET_SQP          0x00000800 /* ira_target_sqp is set */
2392 #define IRAF_VERIFIED_SRC        0x00001000 /* ira_verified_src set */
2393 #define IRAF_RSVP                 0x00002000 /* RSVP packet for rsvpd */
2394 #define IRAF_MROUTE_TUNNEL_SET   0x00004000 /* From ip_mroute_decap */
2395 #define IRAF_PIM_REGISTER        0x00008000 /* From register_mforward */

2397 #define IRAF_TX_MAC_EXEMPTABLE   0x00010000 /* Allow MAC_EXEMPT readdown */
2398 #define IRAF_TX_SHARED_ADDR      0x00020000 /* Arrived on ALL_ZONES addr */
2399 #define IRAF_ESP_UDP_PORTS       0x00040000 /* NAT-traversal packet */
2400 #define IRAF_NO_HW_CKSUM         0x00080000 /* Force software cksum */

2402 #define IRAF_ICMP_ERROR           0x00100000 /* Send to conn_recvicmp */
2403 #define IRAF_ROUTER_ALERT        0x00200000 /* IPv6 router alert */
2404 #define IRAF_L2SRC_SET           0x00400000 /* ira_l2src has been set */
2405 #define IRAF_L2SRC_LOOPBACK      0x00800000 /* Came from us */

2407 #define IRAF_L2DST_MULTICAST     0x01000000 /* Multicast at L2 */
2408 #define IRAF_L2DST_BROADCAST     0x02000000 /* Broadcast at L2 */
2409 /* Unused 0x04000000 */
2410 /* Unused 0x08000000 */

2412 /* Below starts with 0x10000000 */
2413 #define IRAF_IS_IPV4              IAF_IS_IPV4
2414 #define IRAF_TRUSTED_ICMP        IAF_TRUSTED_ICMP
2415 #define IRAF_NO_LOOP_ZONEID_SET  IAF_NO_LOOP_ZONEID_SET
2416 #define IRAF_LOOPBACK_COPY       IAF_LOOPBACK_COPY

2418 /*
2419  * Normally these fields do not have a hold. But in some cases they do, for
2420  * instance when we've gone through ip_*_attr_to/from_mblk.
2421  * We use ira_free_flags to indicate that they have a hold and need to be
2422  * released on cleanup.
2423  */
2424 #define IRA_FREE_CRED              0x00000001 /* ira_cred needs to be rele */
2425 #define IRA_FREE_TSL              0x00000002 /* ira_tsl needs to be rele */

2427 /*
2428  * Optional destination cache entry for path MTU information,
2429  * and ULP metrics.
2430  */
2431 struct dce_s {
2432     uint_t      dce_generation; /* Changed since cached? */
2433     uint_t      dce_flags;      /* See below */
2434     uint_t      dce_ipversion; /* IPv4/IPv6 version */
2435     uint32_t    dce_pmtu;      /* Path MTU if DCEF_PMTU */
2436     uint32_t    dce_ident;     /* Per destination IP ident. */
2437     iulp_t      dce_uinfo;     /* Metrics if DCEF_UINFO */

```

```

2439     struct dce_s *dce_next;
2440     struct dce_s **dce_ptpn;
2441     struct dcb_s *dce_bucket;

2443     union {
2444         in6_addr_t      dceu_v6addr;
2445         ipaddr_t        dceu_v4addr;
2446     } dce_u;
2447 #define dce_v4addr      dce_u.dceu_v4addr
2448 #define dce_v6addr      dce_u.dceu_v6addr
2449 /* Note that for IPv6+IPMP we use the ifindex for the upper interface */
2450 uint_t      dce_ifindex; /* For IPv6 link-locals */

2452     kmutex_t      dce_lock;
2453     uint_t        dce_refcnt;
2454     uint64_t      dce_last_change_time; /* Path MTU. In seconds */

2456     ip_stack_t    *dce_ipst; /* Does not have a netstack_hold */
2457 };

2459 /*
2460  * Values for dce_generation.
2461  * If a DCE has DCE_GENERATION_CONDEMNED, the last dce_refle should delete
2462  * it.
2463  *
2464  * DCE_GENERATION_VERIFY is never stored in dce_generation but it is
2465  * stored in places that cache DCE (such as ixa_dce_generation).
2466  * It is used as a signal that the cache is stale and needs to be reverified.
2467  */
2468 #define DCE_GENERATION_CONDEMNED      0
2469 #define DCE_GENERATION_VERIFY         1
2470 #define DCE_GENERATION_INITIAL        2
2471 #define DCE_IS_CONDEMNED(dce) \
2472     ((dce)->dce_generation == DCE_GENERATION_CONDEMNED)

2476 /*
2477  * Values for ips_src_generation.
2478  *
2479  * SRC_GENERATION_VERIFY is never stored in ips_src_generation but it is
2480  * stored in places that cache IRES (ixa_src_generation). It is used as a
2481  * signal that the cache is stale and needs to be reverified.
2482  */
2483 #define SRC_GENERATION_VERIFY         0
2484 #define SRC_GENERATION_INITIAL        1

2486 /*
2487  * The kernel stores security attributes of all gateways in a database made
2488  * up of one or more tsol_gcdb_t elements. Each tsol_gcdb_t contains the
2489  * security-related credentials of the gateway. More than one gateways may
2490  * share entries in the database.
2491  *
2492  * The tsol_gc_t structure represents the gateway to credential association,
2493  * and refers to an entry in the database. One or more tsol_gc_t entities are
2494  * grouped together to form one or more tsol_gcgrp_t, each representing the
2495  * list of security attributes specific to the gateway. A gateway may be
2496  * associated with at most one credentials group.
2497  */
2498 struct tsol_gcgrp_s;

2500 extern uchar_t ip6opt_ls; /* TX IPv6 enabler */

2502 /*
2503  * Gateway security credential record.

```

```

2504 */
2505 typedef struct tsol_gcdb_s {
2506     uint_t          gcdb_refcnt;    /* reference count */
2507     struct rtlsa_s  gcdb_attr;      /* security attributes */
2508 #define gcdb_mask   gcdb_attr.rtsa_mask
2509 #define gcdb_doi    gcdb_attr.rtsa_doi
2510 #define gcdb_slrange gcdb_attr.rtsa_slrange
2511 } tsol_gcdb_t;

2513 /*
2514 * Gateway to credential association.
2515 */
2516 typedef struct tsol_gc_s {
2517     uint_t          gc_refcnt;      /* reference count */
2518     struct tsol_gcgrp_s *gc_grp;    /* pointer to group */
2519     struct tsol_gc_s *gc_prev;      /* previous in list */
2520     struct tsol_gc_s *gc_next;      /* next in list */
2521     tsol_gcdb_t     *gc_db;         /* pointer to actual credentials */
2522 } tsol_gc_t;

2524 /*
2525 * Gateway credentials group address.
2526 */
2527 typedef struct tsol_gcgrp_addr_s {
2528     int             ga_af;          /* address family */
2529     in6_addr_t      ga_addr;        /* IPv4 mapped or IPv6 address */
2530 } tsol_gcgrp_addr_t;

2532 /*
2533 * Gateway credentials group.
2534 */
2535 typedef struct tsol_gcgrp_s {
2536     uint_t          gcgrp_refcnt;    /* reference count */
2537     krwlock_t       gcgrp_rwlock;    /* lock to protect following */
2538     uint_t          gcgrp_count;      /* number of credentials */
2539     tsol_gc_t       *gcgrp_head;     /* first credential in list */
2540     tsol_gc_t       *gcgrp_tail;     /* last credential in list */
2541     tsol_gcgrp_addr_t gcgrp_addr;    /* next-hop gateway address */
2542 } tsol_gcgrp_t;

2544 extern kmutex_t gcgrp_lock;

2546 #define GC_REFRELE(p) { \
2547     ASSERT((p)->gc_grp != NULL); \
2548     rw_enter(&(p)->gc_grp->gcgrp_rwlock, RW_WRITER); \
2549     ASSERT((p)->gc_refcnt > 0); \
2550     if (--((p)->gc_refcnt) == 0) \
2551         gc_inactive(p); \
2552     else \
2553         rw_exit(&(p)->gc_grp->gcgrp_rwlock); \
2554 }

2556 #define GCGRP_REFHOLD(p) { \
2557     mutex_enter(&gcgrp_lock); \
2558     ++((p)->gcgrp_refcnt); \
2559     ASSERT((p)->gcgrp_refcnt != 0); \
2560     mutex_exit(&gcgrp_lock); \
2561 }

2563 #define GCGRP_REFRELE(p) { \
2564     mutex_enter(&gcgrp_lock); \
2565     ASSERT((p)->gcgrp_refcnt > 0); \
2566     if (--((p)->gcgrp_refcnt) == 0) \
2567         gcgrp_inactive(p); \
2568     ASSERT(MUTEX_HELD(&gcgrp_lock)); \
2569     mutex_exit(&gcgrp_lock); \

```

```

2570 }

2572 /*
2573 * IRE gateway security attributes structure, pointed to by tsol_ire_gw_secattr
2574 */
2575 struct tsol_tnrhc;

2577 struct tsol_ire_gw_secattr_s {
2578     kmutex_t        igsa_lock;      /* lock to protect following */
2579     struct tsol_tnrhc *igsa_rhc;    /* host entry for gateway */
2580     tsol_gc_t       *igsa_gc;       /* for prefix IREs */
2581 };

2583 void irb_refrele_ftable(irb_t *);

2585 extern struct kmem_cache *rt_entry_cache;

2587 typedef struct ire4 {
2588     ipaddr_t        ire4_mask;      /* Mask for matching this IRE. */
2589     ipaddr_t        ire4_addr;      /* Address this IRE represents. */
2590     ipaddr_t        ire4_gateway_addr; /* Gateway including for IRE_ONLINK */
2591     ipaddr_t        ire4_setsrc_addr; /* RTF_SETSRC */
2592 } ire4_t;

2594 typedef struct ire6 {
2595     in6_addr_t      ire6_mask;      /* Mask for matching this IRE. */
2596     in6_addr_t      ire6_addr;      /* Address this IRE represents. */
2597     in6_addr_t      ire6_gateway_addr; /* Gateway including for IRE_ONLINK */
2598     in6_addr_t      ire6_setsrc_addr; /* RTF_SETSRC */
2599 } ire6_t;

2601 typedef union ire_addr {
2602     ire6_t          ire6_u;
2603     ire4_t          ire4_u;
2604 } ire_addr_u_t;

2606 /*
2607 * Internet Routing Entry
2608 * When we have multiple identical IREs we logically add them by manipulating
2609 * ire_identical_ref and ire_delete first decrements
2610 * that and when it reaches 1 we know it is the last IRE.
2611 * "identical" is defined as being the same for:
2612 * ire_addr, ire_netmask, ire_gateway, ire_ill, ire_zoneid, and ire_type
2613 * For instance, multiple IRE_BROADCASTs for the same subnet number are
2614 * viewed as identical, and so are the IRE_INTERFACES when there are
2615 * multiple logical interfaces (on the same ill) with the same subnet prefix.
2616 */
2617 struct ire_s {
2618     struct ire_s    *ire_next;      /* The hash chain must be first. */
2619     struct ire_s    **ire_ptpn;     /* Pointer to previous next. */
2620     uint32_t        ire_refcnt;      /* Number of references */
2621     ill_t           *ire_ill;
2622     uint32_t        ire_identical_ref; /* IRE_INTERFACE, IRE_BROADCAST */
2623     uchar_t         ire_ipversion;  /* IPv4/IPv6 version */
2624     ushort_t        ire_type;       /* Type of IRE */
2625     uint_t          ire_generation; /* Generation including CONDEMNED */
2626     uint_t          ire_ib_pkt_count; /* Inbound packets for ire_addr */
2627     uint_t          ire_ob_pkt_count; /* Outbound packets to ire_addr */
2628     time_t          ire_create_time; /* Time (in secs) IRE was created. */
2629     uint32_t        ire_flags;      /* flags related to route (RTF_*) */
2630     /*
2631      * ire_testhidden is TRUE for INTERFACE IREs of IS_UNDER_IPMP(ill)
2632      * interfaces
2633      */
2634     boolean_t       ire_testhidden;
2635     pfirecv_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  */

```

```

2768 * Note: This private data structure and related IPPF_* constant
2769 * definitions are exposed to enable compilation of some debugging tools
2770 * like lsof which use struct tcp_t in <inet/tcp.h>. This is intended to be
2771 * a temporary hack and long term alternate interfaces should be defined
2772 * to support the needs of such tools and private definitions moved to
2773 * private headers.
2774 */
2775 struct ip_pkt_s {
2776     uint_t          ipp_fields;          /* Which fields are valid */
2777     in6_addr_t      ipp_addr;           /* pktinfo src/dst addr */
2778 #define ipp_addr_v4 V4_PART_OF_V6(ipp_addr)
2779     uint_t          ipp_unicast_hops;    /* IPV6_UNICAST_HOPS, IP_TTL */
2780     uint_t          ipp_hoplimit;       /* IPV6_HOPLIMIT */
2781     uint_t          ipp_hopoptslen;
2782     uint_t          ipp_rthdrdstoptslen;
2783     uint_t          ipp_rthdrhlen;
2784     uint_t          ipp_dstoptslen;
2785     uint_t          ipp_fraghdrhlen;
2786     ip6_hbh_t       ipp_hopopts;
2787     ip6_dest_t      ipp_rthdrdstopts;
2788     ip6_rthdr_t     ipp_rthdr;
2789     ip6_dest_t      ipp_dstopts;
2790     ip6_frag_t      ipp_fraghdr;
2791     uint8_t         ipp_tclass;         /* IPV6_TCLASS */
2792     uint8_t         ipp_type_of_service; /* IP_TOS */
2793     uint_t          ipp_ipv4_options_len; /* Len of IPv4 options */
2794     uint8_t         ipp_ipv4_options;   /* Ptr to IPv4 options */
2795     uint_t          ipp_label_len_v4;   /* Len of TX label for IPv4 */
2796     uint8_t         ipp_label_v4;      /* TX label for IPv4 */
2797     uint_t          ipp_label_len_v6;   /* Len of TX label for IPv6 */
2798     uint8_t         ipp_label_v6;      /* TX label for IPv6 */
2799 };
2800 typedef struct ip_pkt_s ip_pkt_t;

2802 extern void ip_pkt_free(ip_pkt_t *); /* free storage inside ip_pkt_t */
2803 extern ipaddr_t ip_pkt_source_route_v4(const ip_pkt_t *);
2804 extern in6_addr_t *ip_pkt_source_route_v6(const ip_pkt_t *);
2805 extern int ip_pkt_copy(ip_pkt_t *, ip_pkt_t *, int);
2806 extern void ip_pkt_source_route_reverse_v4(ip_pkt_t *);

2808 /* ipp_fields values */
2809 #define IPPF_ADDR 0x0001 /* Part of in6_pktinfo: src/dst addr */
2810 #define IPPF_HOPLIMIT 0x0002 /* Overrides unicast and multicast */
2811 #define IPPF_TCLASS 0x0004 /* Overrides class in sin6_flowinfo */

2813 #define IPPF_HOPOPTS 0x0010 /* ipp_hopopts set */
2814 #define IPPF_RTHDR 0x0020 /* ipp_rthdr set */
2815 #define IPPF_RTHDRDSTOPTS 0x0040 /* ipp_rthdrdstopts set */
2816 #define IPPF_DSTOPTS 0x0080 /* ipp_dstopts set */

2818 #define IPPF_IPV4_OPTIONS 0x0100 /* ipp_ipv4_options set */
2819 #define IPPF_LABEL_V4 0x0200 /* ipp_label_v4 set */
2820 #define IPPF_LABEL_V6 0x0400 /* ipp_label_v6 set */

2822 #define IPPF_FRAGHDR 0x0800 /* Used for IPsec receive side */

2824 /*
2825 * Data structure which is passed to conn_opt_get/set.
2826 * The conn_t is included even though it can be inferred from queue_t.
2827 * setsockopt and getsockopt use conn_ixa and conn_xmit_ipp. However,
2828 * when handling ancillary data we use separate ixa and ipps.
2829 */
2830 typedef struct conn_opt_arg_s {
2831     conn_t *coa_connp;
2832     ip_xmit_attr_t *coa_ixa;
2833     ip_pkt_t *coa_ipp;

```

```

2834     boolean_t coa_ancillary; /* Ancillary data and not setsockopt */
2835     uint_t coa_changed; /* See below */
2836 } conn_opt_arg_t;

2838 /*
2839 * Flags for what changed.
2840 * If we want to be more efficient in the future we can have more fine
2841 * grained flags e.g., a flag for just IP_TOS changing.
2842 * For now we either call ip_set_destination (for "route changed")
2843 * and/or conn_build_hdr_template/conn_prepend_hdr (for "header changed").
2844 */
2845 #define COA_HEADER_CHANGED 0x0001
2846 #define COA_ROUTE_CHANGED 0x0002
2847 #define COA_RCVBUF_CHANGED 0x0004 /* SO_RCVBUF */
2848 #define COA_SNDBUF_CHANGED 0x0008 /* SO_SNDBUF */
2849 #define COA_WROFF_CHANGED 0x0010 /* Header size changed */
2850 #define COA_ICMP_BIND_NEEDED 0x0020
2851 #define COA_OOINLINE_CHANGED 0x0040

2853 #define TCP_PORTS_OFFSET 0
2854 #define UDP_PORTS_OFFSET 0

2856 /*
2857 * lookups return the ill/ipif only if the flags are clear OR Iam writer.
2858 * ill / ipif lookup functions increment the refcnt on the ill / ipif only
2859 * after calling these macros. This ensures that the refcnt on the ipif or
2860 * ill will eventually drop down to zero.
2861 */
2862 #define ILL_LOOKUP_FAILED 1 /* Used as error code */
2863 #define IPIF_LOOKUP_FAILED 2 /* Used as error code */

2865 #define ILL_CAN_LOOKUP(ill) \
2866     (!((ill)->ill_state_flags & ILL_CONDEMNED) || \
2867     IAM_WRITER_ILL(ill))

2869 #define ILL_IS_CONDEMNED(ill) \
2870     ((ill)->ill_state_flags & ILL_CONDEMNED)

2872 #define IPIF_CAN_LOOKUP(ipif) \
2873     (!((ipif)->ipif_state_flags & IPIF_CONDEMNED) || \
2874     IAM_WRITER_IPIF(ipif))

2876 #define IPIF_IS_CONDEMNED(ipif) \
2877     ((ipif)->ipif_state_flags & IPIF_CONDEMNED)

2879 #define IPIF_IS_CHANGING(ipif) \
2880     ((ipif)->ipif_state_flags & IPIF_CHANGING)

2882 /* Macros used to assert that this thread is a writer */
2883 #define IAM_WRITER_IPSQ(ipsq) ((ipsq)->ipsq_xop->ipx_writer == curthread)
2884 #define IAM_WRITER_ILL(ill) IAM_WRITER_IPSQ((ill)->ill_phyint->phyint_ipsq)
2885 #define IAM_WRITER_IPIF(ipif) IAM_WRITER_ILL((ipif)->ipif_ill)

2887 /*
2888 * Grab ill locks in the proper order. The order is highest addressed
2889 * ill is locked first.
2890 */
2891 #define GRAB_ILL_LOCKS(ill_1, ill_2) \
2892 { \
2893     if ((ill_1) > (ill_2)) { \
2894         if (ill_1 != NULL) \
2895             mutex_enter(&(ill_1)->ill_lock); \
2896         if (ill_2 != NULL) \
2897             mutex_enter(&(ill_2)->ill_lock); \
2898     } else { \
2899         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]lifreq 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_oval
2961 #define ips_ip_ire_reclaim_fraction ips_propinfo_tbl[8].prop_cur_oval
2962 #define ips_ip_nce_reclaim_fraction ips_propinfo_tbl[9].prop_cur_oval
2963 #define ips_ip_dce_reclaim_fraction ips_propinfo_tbl[10].prop_cur_oval
2964 #define ips_ip_def_ttl ips_propinfo_tbl[11].prop_cur_oval
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_oval
2967 #define ips_ip_pathmtu_interval ips_propinfo_tbl[14].prop_cur_oval
2968 #define ips_ip_icmp_return ips_propinfo_tbl[15].prop_cur_oval
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_oval
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_oval
2974 #define ips_ip_icmp_err_interval ips_propinfo_tbl[21].prop_cur_oval
2975 #define ips_ip_icmp_err_burst ips_propinfo_tbl[22].prop_cur_oval
2976 #define ips_ip_reass_queue_bytes ips_propinfo_tbl[23].prop_cur_oval
2977 #define ips_ip_strict_dst_multihoming ips_propinfo_tbl[24].prop_cur_oval
2978 #define ips_ip_addrs_per_if ips_propinfo_tbl[25].prop_cur_oval
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_oval
2985 #define ips_max_unicast_solicit ips_propinfo_tbl[30].prop_cur_oval
2986 #define ips_ipv6_def_hops ips_propinfo_tbl[31].prop_cur_oval
2987 #define ips_ipv6_icmp_return ips_propinfo_tbl[32].prop_cur_oval
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_oval
2993 #define ips_src_check ips_propinfo_tbl[38].prop_cur_oval
2994 #define ips_ipsec_policy_log_interval ips_propinfo_tbl[39].prop_cur_oval
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_oval
2997 #define ips_ip_ndp_unsolicit_count ips_propinfo_tbl[42].prop_cur_oval
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_oval
3002 #define ips_ip_ecmp_behavior ips_propinfo_tbl[45].prop_cur_oval
3003 #define ips_ip_multirt_ttl ips_propinfo_tbl[46].prop_cur_oval
3004 #define ips_ip_ire_badcnt_lifetime ips_propinfo_tbl[47].prop_cur_oval
3005 #define ips_ip_max_temp_idle ips_propinfo_tbl[48].prop_cur_oval
3006 #define ips_ip_max_temp_defend ips_propinfo_tbl[49].prop_cur_oval
3007 #define ips_ip_max_defend ips_propinfo_tbl[50].prop_cur_oval
3008 #define ips_ip_defend_interval ips_propinfo_tbl[51].prop_cur_oval
3009 #define ips_ip_dup_recovery ips_propinfo_tbl[52].prop_cur_oval
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_oval
3013 #define ips_mld_max_version ips_propinfo_tbl[56].prop_cur_oval
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_oval
3017 #define ips_ipv6_reassembly_timeout ips_propinfo_tbl[60].prop_cur_oval
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_oval
3020 #define ips_arp_fastprobe_delay ips_propinfo_tbl[63].prop_cur_oval
3021 #define ips_arp_probe_interval ips_propinfo_tbl[64].prop_cur_oval
3022 #define ips_arp_fastprobe_interval ips_propinfo_tbl[65].prop_cur_oval
3023 #define ips_arp_probe_count ips_propinfo_tbl[66].prop_cur_oval
3024 #define ips_arp_fastprobe_count ips_propinfo_tbl[67].prop_cur_oval
3025 #define ips_ipv4_dad_announce_interval ips_propinfo_tbl[68].prop_cur_oval
3026 #define ips_ipv6_dad_announce_interval ips_propinfo_tbl[69].prop_cur_oval
3027 #define ips_arp_defend_interval ips_propinfo_tbl[70].prop_cur_oval
3028 #define ips_arp_defend_rate ips_propinfo_tbl[71].prop_cur_oval
3029 #define ips_ndp_defend_interval ips_propinfo_tbl[72].prop_cur_oval
3030 #define ips_ndp_defend_rate ips_propinfo_tbl[73].prop_cur_oval
3031 #define ips_arp_defend_period ips_propinfo_tbl[74].prop_cur_oval

```

```

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 macros used inside of ip
3074  * The callers use the above INTERESTED macros first, hence
3075  * the he_interested check is superfluous.
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 */

```

```

3164 #define CLASSD(addr)      (((addr) & N_IN_CLASSD_NET) == N_INADDR_UNSPEC_GROUP)
3165 #define CLASSE(addr)      (((addr) & N_IN_CLASSD_NET) == N_IN_CLASSD_NET)
3166 #define IP_LOOPBACK_ADDR(addr) \
3167     (((addr) & N_IN_CLASSA_NET) == N_IN_LOOPBACK_NET))

3169 extern int      ip_debug;
3170 extern uint_t  ip_thread_data;
3171 extern krwlock_t ip_thread_rwlock;
3172 extern list_t  ip_thread_list;

3174 #ifdef IP_DEBUG
3175 #include <sys/debug.h>
3176 #include <sys/promif.h>

3178 #define ip0dbg(a)          printf a
3179 #define ip1dbg(a)          if (ip_debug > 2) printf a
3180 #define ip2dbg(a)          if (ip_debug > 3) printf a
3181 #define ip3dbg(a)          if (ip_debug > 4) printf a
3182 #else
3183 #define ip0dbg(a)          /* */
3184 #define ip1dbg(a)          /* */
3185 #define ip2dbg(a)          /* */
3186 #define ip3dbg(a)          /* */
3187 #endif /* IP_DEBUG */

3189 /* Default MAC-layer address string length for mac_colon_addr */
3190 #define MAC_STR_LEN      128

3192 struct mac_header_info_s;

3194 extern void      ill_frag_timer(void *);
3195 extern ill_t    *ill_first(int, int, ill_walk_context_t *, ip_stack_t *);
3196 extern ill_t    *ill_next(ill_walk_context_t *, ill_t *);
3197 extern void      ill_frag_timer_start(ill_t *);
3198 extern void      ill_nic_event_dispatch(ill_t *, lif_if_t, nic_event_t,
3199     nic_event_data_t, size_t);
3200 extern mblk_t   *ip_carve_mp(mblk_t **, ssize_t);
3201 extern mblk_t   *ip_dlpi_alloc(size_t, t_uscalar_t);
3202 extern mblk_t   *ip_dlnotify_alloc(uint_t, uint_t);
3203 extern mblk_t   *ip_dlnotify_alloc2(uint_t, uint_t, uint_t);
3204 extern char     *ip_dot_addr(ipaddr_t, char *);
3205 extern const char *mac_colon_addr(const uint8_t *, size_t, char *, size_t);
3206 extern void      ip_lwput(queue_t *, mblk_t *);
3207 extern boolean_t icmp_err_rate_limit(ip_stack_t *);
3208 extern void      icmp_frag_needed(mblk_t *, int, ip_rcv_attr_t *);
3209 extern mblk_t   *icmp_inbound_v4(mblk_t *, ip_rcv_attr_t *);
3210 extern void      icmp_time_exceeded(mblk_t *, uint8_t, ip_rcv_attr_t *);
3211 extern void      icmp_unreachable(mblk_t *, uint8_t, ip_rcv_attr_t *);
3212 extern boolean_t ip_ipsec_policy_inherit(conn_t *, conn_t *, ip_rcv_attr_t *);
3213 extern void      *ip_pullup(mblk_t *, ssize_t, ip_rcv_attr_t *);
3214 extern void      ip_setl2src(mblk_t *, ip_rcv_attr_t *, ill_t *);
3215 extern mblk_t   *ip_check_and_align_header(mblk_t *, uint_t, ip_rcv_attr_t *);
3216 extern mblk_t   *ip_check_length(mblk_t *, uchar_t *, ssize_t, uint_t, uint_t,
3217     ip_rcv_attr_t *);
3218 extern mblk_t   *ip_check_optlen(mblk_t *, ipha_t *, uint_t, uint_t,
3219     ip_rcv_attr_t *);
3220 extern mblk_t   *ip_fix_dbref(mblk_t *, ip_rcv_attr_t *);
3221 extern uint_t   ip_cksum(mblk_t *, int, uint32_t);
3222 extern int      ip_close(queue_t *, int);
3223 extern uint16_t ip_csum_hdr(ipha_t *);
3224 extern void      ip_forward_xmit_v4(nce_t *, ill_t *, mblk_t *, ipha_t *,
3225     ip_rcv_attr_t *, uint32_t, uint32_t);
3226 extern boolean_t ip_forward_options(mblk_t *, ipha_t *, ill_t *,
3227     ip_rcv_attr_t *);
3228 extern int      ip_fragment_v4(mblk_t *, nce_t *, iaflags_t, uint_t, uint32_t,
3229     uint32_t, zoneid_t, zoneid_t, pfirepostfrag_t postfragfn,

```

```

3230     uintptr_t *cookie);
3231 extern void      ip_proto_not_sup(mblk_t *, ip_rcv_attr_t *);
3232 extern void      ip_ire_g_fini(void);
3233 extern void      ip_ire_g_init(void);
3234 extern void      ip_ire_fini(ip_stack_t *);
3235 extern void      ip_ire_init(ip_stack_t *);
3236 extern void      ip_mdata_to_mhi(ill_t *, mblk_t *, struct mac_header_info_s *);
3237 extern int      ip_openv4(queue_t *q, dev_t *devp, int flag, int sflag,
3238     cred_t *credp);
3239 extern int      ip_openv6(queue_t *q, dev_t *devp, int flag, int sflag,
3240     cred_t *credp);
3241 extern int      ip_reassemble(mblk_t *, ipf_t *, uint_t, boolean_t, ill_t *,
3242     size_t);
3243 extern void      ip_rput(queue_t *, mblk_t *);
3244 extern void      ip_input(ill_t *, ill_rx_ring_t *, mblk_t *,
3245     struct mac_header_info_s *);
3246 extern void      ip_input_v6(ill_t *, ill_rx_ring_t *, mblk_t *,
3247     struct mac_header_info_s *);
3248 extern mblk_t   *ip_input_common_v4(ill_t *, ill_rx_ring_t *, mblk_t *,
3249     struct mac_header_info_s *, queue_t *, mblk_t **, uint_t *);
3250 extern mblk_t   *ip_input_common_v6(ill_t *, ill_rx_ring_t *, mblk_t *,
3251     struct mac_header_info_s *, queue_t *, mblk_t **, uint_t *);
3252 extern void      ill_input_full_v4(mblk_t *, void *, void *,
3253     ip_rcv_attr_t *, rtc_t *);
3254 extern void      ill_input_short_v4(mblk_t *, void *, void *,
3255     ip_rcv_attr_t *, rtc_t *);
3256 extern void      ill_input_full_v6(mblk_t *, void *, void *,
3257     ip_rcv_attr_t *, rtc_t *);
3258 extern void      ill_input_short_v6(mblk_t *, void *, void *,
3259     ip_rcv_attr_t *, rtc_t *);
3260 extern ipaddr_t ip_input_options(ipha_t *, ipaddr_t, mblk_t *,
3261     ip_rcv_attr_t *, int *);
3262 extern boolean_t ip_input_local_options(mblk_t *, ipha_t *, ip_rcv_attr_t *);
3263 extern mblk_t   *ip_input_fragment(mblk_t *, ipha_t *, ip_rcv_attr_t *);
3264 extern mblk_t   *ip_input_fragment_v6(mblk_t *, ip6_t *, ip6_frag_t *, uint_t,
3265     ip_rcv_attr_t *);
3266 extern void      ip_input_post_ipsec(mblk_t *, ip_rcv_attr_t *);
3267 extern void      ip_fanout_v4(mblk_t *, ipha_t *, ip_rcv_attr_t *);
3268 extern void      ip_fanout_v6(mblk_t *, ip6_t *, ip_rcv_attr_t *);
3269 extern void      ip_fanout_proto_conn(conn_t *, mblk_t *, ipha_t *, ip6_t *,
3270     ip_rcv_attr_t *);
3271 extern void      ip_fanout_proto_v4(mblk_t *, ipha_t *, ip_rcv_attr_t *);
3272 extern void      ip_fanout_send_icmp_v4(mblk_t *, uint_t, uint_t,
3273     ip_rcv_attr_t *);
3274 extern void      ip_fanout_udp_conn(conn_t *, mblk_t *, ipha_t *, ip6_t *,
3275     ip_rcv_attr_t *);
3276 extern void      ip_fanout_udp_multi_v4(mblk_t *, ipha_t *, uint16_t, uint16_t,
3277     ip_rcv_attr_t *);
3278 extern mblk_t   *zero_spi_check(mblk_t *, ip_rcv_attr_t *);
3279 extern void      ip_build_hdrs_v4(uchar_t *, uint_t, const ip_pkt_t *, uint8_t);
3280 extern int      ip_find_hdr_v4(ipha_t *, ip_pkt_t *, boolean_t);
3281 extern int      ip_total_hdrs_len_v4(const ip_pkt_t *);

3283 extern mblk_t   *ip_accept_tcp(ill_t *, ill_rx_ring_t *, queue_t *,
3284     mblk_t **, uint_t *cnt);
3285 extern void      ip_rput_dlpi(ill_t *, mblk_t *);
3286 extern void      ip_rput_notdata(ill_t *, mblk_t *);

3288 extern void      ip_mib2_add_ip_stats(mib2_ipIfStatsEntry_t *,
3289     mib2_ipIfStatsEntry_t *);
3290 extern void      ip_mib2_add_icmp6_stats(mib2_ipv6IfIcmpEntry_t *,
3291     mib2_ipv6IfIcmpEntry_t *);
3292 extern void      ip_rput_other(ipsq_t *, queue_t *, mblk_t *, void *);
3293 extern ire_t    *ip_check_mulihome(void *, ire_t *, ill_t *);
3294 extern void      ip_send_potential_redirect_v4(mblk_t *, ipha_t *, ire_t *,
3295     ip_rcv_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_rcv_attr_replace_label(ip_rcv_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_rcv_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_rcv_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_message_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 iprintiv6;

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_ncec(ncec_t *);
3418 extern void ipmp_ncec_refresh_ncec(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_rcv_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 #ifndef 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

```

```

3560 * include mac/dls header files here.
3561 */
3562 typedef boolean_t      (*ip_mac_intr_disable_t)(void *);
3563 typedef void           (*ip_mac_intr_enable_t)(void *);
3564 typedef ip_mac_tx_cookie_t (*ip_dld_tx_t)(void *, mblk_t *,
3565     uint64_t, uint16_t);
3566 typedef void           (*ip_flow_enable_t)(void *, ip_mac_tx_cookie_t);
3567 typedef void           (*ip_dld_callb_t)(void *,
3568     ip_flow_enable_t, void *);
3569 typedef boolean_t      (*ip_dld_fctl_t)(void *, ip_mac_tx_cookie_t);
3570 typedef int            (*ip_capab_func_t)(void *, uint_t,
3571     void *, uint_t);

3573 /*
3574 * POLLING README
3575 * sq_get_pkts() is called to pick packets from softing in poll mode. It
3576 * calls rr_rx to get the chain and process it with rr_ip_accept.
3577 * rr_rx = mac_soft_ring_poll() to pick packets
3578 * rr_ip_accept = ip_accept_tcp() to process packets
3579 */

3581 /*
3582 * XXX: With protocol, service specific squeues, they will have
3583 * specific acceptor functions.
3584 */
3585 typedef mblk_t *(*ip_mac_rx_t)(void *, size_t);
3586 typedef mblk_t *(*ip_accept_t)(ill_t *, ill_rx_ring_t *,
3587     squeue_t *, mblk_t *, mblk_t **, uint_t *);

3589 /*
3590 * rr_intr_enable, rr_intr_disable, rr_rx_handle, rr_rx:
3591 * May be accessed while in the squeue AND after checking that SQS_POLL_CAPAB
3592 * is set.
3593 *
3594 * rr_ring_state: Protected by ill_lock.
3595 */
3596 struct ill_rx_ring {
3597     ip_mac_intr_disable_t  rr_intr_disable; /* Interrupt disabling func */
3598     ip_mac_intr_enable_t  rr_intr_enable; /* Interrupt enabling func */
3599     void                   *rr_intr_handle; /* Handle interrupt funcs */
3600     ip_mac_rx_t            rr_rx; /* Driver receive function */
3601     ip_accept_t            rr_ip_accept; /* IP accept function */
3602     void                   *rr_rx_handle; /* Handle for Rx ring */
3603     squeue_t               *rr_sq; /* Squeue the ring is bound to */
3604     ill_t                  *rr_ill; /* back pointer to ill */
3605     ip_ring_state_t        rr_ring_state; /* State of this ring */
3606 };

3608 /*
3609 * IP - DLD direct function call capability
3610 * Suffixes, df - dld function, dh - dld handle,
3611 * cf - client (IP) function, ch - client handle
3612 */
3613 typedef struct ill_dld_direct_s {
3614     ip_dld_tx_t            idd_tx_df; /* str_mdata_fastpath_put */
3615     void                   *idd_tx_dh; /* dld_str_t *dsp */
3616     ip_dld_callb_t        idd_tx_cb_df; /* mac_tx_srs_notify */
3617     void                   *idd_tx_cb_dh; /* mac_client_handle_t *mch */
3618     ip_dld_fctl_t         idd_tx_fctl_df; /* mac_tx_is_flow_blocked */
3619     void                   *idd_tx_fctl_dh; /* mac_client_handle */
3620 } ill_dld_direct_t;

3622 /* IP - DLD polling capability */
3623 typedef struct ill_dld_poll_s {
3624     ill_rx_ring_t         idp_ring_tbl[ILL_MAX_RINGS];
3625 } ill_dld_poll_t;

```

```

3627 /* Describes ill->ill_dld_capab */
3628 struct ill_dld_capab_s {
3629     ip_capab_func_t      idc_capab_df; /* dld_capab_func */
3630     void                  *idc_capab_dh; /* dld_str_t *dsp */
3631     ill_dld_direct_t     idc_direct;
3632     ill_dld_poll_t       idc_poll;
3633 };

3635 /*
3636 * IP squeues exports
3637 */
3638 extern boolean_t        ip_squeue_fanout;

3640 #define IP_SQUEUE_GET(hint) ip_squeue_random(hint)

3642 extern void ip_squeue_init(void (*)(squeue_t *));
3643 extern squeue_t *ip_squeue_random(uint_t);
3644 extern squeue_t *ip_squeue_get(ill_rx_ring_t *);
3645 extern squeue_t *ip_squeue_getfree(priority);
3646 extern int ip_squeue_cpu_move(squeue_t *, processorid_t);
3647 extern void *ip_squeue_add_ring(ill_t *, void *);
3648 extern void ip_squeue_bind_ring(ill_t *, ill_rx_ring_t *, processorid_t);
3649 extern void ip_squeue_clean_ring(ill_t *, ill_rx_ring_t *);
3650 extern void ip_squeue_quiesce_ring(ill_t *, ill_rx_ring_t *);
3651 extern void ip_squeue_restart_ring(ill_t *, ill_rx_ring_t *);
3652 extern void ip_squeue_clean_all(ill_t *);
3653 extern boolean_t        ip_source_routed(ipha_t *, ip_stack_t *);

3655 extern void tcp_wput(queue_t *, mblk_t *);

3657 extern int ip_fill_mtuserinfo(conn_t *, ip_xmit_attr_t *,
3658     struct ip6_mtuserinfo *);
3659 extern hook_t *ipobs_register_hook(netstack_t *, pfv_t);
3660 extern void ipobs_unregister_hook(netstack_t *, hook_t *);
3661 extern void ipobs_hook(mblk_t *, int, zoneid_t, zoneid_t, const ill_t *,
3662     ip_stack_t *);
3663 typedef void (*ipsq_func_t)(ipsq_t *, queue_t *, mblk_t *, void *);

3665 extern void dce_g_init(void);
3666 extern void dce_g_destroy(void);
3667 extern void dce_stack_init(ip_stack_t *);
3668 extern void dce_stack_destroy(ip_stack_t *);
3669 extern void dce_cleanup(uint_t, ip_stack_t *);
3670 extern void *dce_get_default(ip_stack_t *);
3671 extern dce_t *dce_lookup_pkt(mblk_t *, ip_xmit_attr_t *, uint_t *);
3672 extern dce_t *dce_lookup_v4(ipaddr_t, ip_stack_t *, uint_t *);
3673 extern dce_t *dce_lookup_v6(const in6_addr_t *, uint_t, ip_stack_t *,
3674     uint_t *);
3675 extern dce_t *dce_lookup_and_add_v4(ipaddr_t, ip_stack_t *);
3676 extern dce_t *dce_lookup_and_add_v6(const in6_addr_t *, uint_t,
3677     ip_stack_t *);
3678 extern int dce_update_uinfo_v4(ipaddr_t, iulp_t *, ip_stack_t *);
3679 extern int dce_update_uinfo_v6(const in6_addr_t *, uint_t, iulp_t *,
3680     ip_stack_t *);
3681 extern int dce_update_uinfo(const in6_addr_t *, uint_t, iulp_t *,
3682     ip_stack_t *);
3683 extern void dce_increment_generation(dce_t *);
3684 extern void dce_increment_all_generations(boolean_t, ip_stack_t *);
3685 extern void dce_refrele(dce_t *);
3686 extern void dce_refhold(dce_t *);
3687 extern void dce_refrele_notr(dce_t *);
3688 extern void dce_refhold_notr(dce_t *);
3689 mblk_t *ip_snmp_get_mib2_ip_dce(queue_t *, mblk_t *, ip_stack_t *ipst);

3691 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_rcv_attr_to_mblk(ip_rcv_attr_t *);
3704 extern boolean_t ip_rcv_attr_from_mblk(mblk_t *, ip_rcv_attr_t *);
3705 extern mblk_t *ip_rcv_attr_free_mblk(mblk_t *);
3706 extern boolean_t ip_rcv_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_CONN_REQ_UNBOUND 47
3760 #endif /* ! codereview */

3762 extern sin_t sin_null; /* Zero address for quick clears */
3763 extern sin6_t sin6_null; /* Zero address for quick clears */

3765 #endif /* _KERNEL */

3767 #ifdef __cplusplus
3768 }
3769 #endif

3771 #endif /* _INET_IP_H */

```

new/usr/src/uts/common/inet/ip/ip.c

1

```
*****
448519 Wed Jun 13 12:04:45 2012
new/usr/src/uts/common/inet/ip/ip.c
%B
*****
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 */
21
22 /*
23  * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 1990 Mentat Inc.
25  * Copyright (c) 2011 Joyent, Inc. All rights reserved.
26  */
27
28 #include <sys/types.h>
29 #include <sys/stream.h>
30 #include <sys/dlpi.h>
31 #include <sys/stropts.h>
32 #include <sys/sysmacros.h>
33 #include <sys/strsubr.h>
34 #include <sys/strlog.h>
35 #include <sys/strsun.h>
36 #include <sys/zone.h>
37 #define _SUN_TPI_VERSION 2
38 #include <sys/tihdr.h>
39 #include <sys/xti_inet.h>
40 #include <sys/ddi.h>
41 #include <sys/suntpi.h>
42 #include <sys/cmn_err.h>
43 #include <sys/debug.h>
44 #include <sys/kobj.h>
45 #include <sys/modctl.h>
46 #include <sys/atomic.h>
47 #include <sys/policy.h>
48 #include <sys/priv.h>
49 #include <sys/taskq.h>
50
51 #include <sys/systm.h>
52 #include <sys/param.h>
53 #include <sys/kmem.h>
54 #include <sys/sdt.h>
55 #include <sys/socket.h>
56 #include <sys/vtrace.h>
57 #include <sys/isa_defs.h>
58 #include <sys/mac.h>
59 #include <net/if.h>
60 #include <net/if_arp.h>
61 #include <net/route.h>
```

new/usr/src/uts/common/inet/ip/ip.c

2

```
62 #include <sys/sockio.h>
63 #include <netinet/in.h>
64 #include <net/if_dl.h>
65
66 #include <inet/common.h>
67 #include <inet/mi.h>
68 #include <inet/mib2.h>
69 #include <inet/nd.h>
70 #include <inet/arp.h>
71 #include <inet/snmpcom.h>
72 #include <inet/optcom.h>
73 #include <inet/kstatcom.h>
74
75 #include <netinet/igmp_var.h>
76 #include <netinet/ip6.h>
77 #include <netinet/icmp6.h>
78 #include <netinet/sctp.h>
79
80 #include <inet/ip.h>
81 #include <inet/ip_impl.h>
82 #include <inet/ip6.h>
83 #include <inet/ip6_asp.h>
84 #include <inet/tcp.h>
85 #include <inet/tcp_impl.h>
86 #include <inet/ip_multi.h>
87 #include <inet/ip_if.h>
88 #include <inet/ip_ire.h>
89 #include <inet/ip_ftable.h>
90 #include <inet/ip_rts.h>
91 #include <inet/ip_ndp.h>
92 #include <inet/ip_listutils.h>
93 #include <netinet/igmp.h>
94 #include <netinet/ip_mroute.h>
95 #include <inet/ipp_common.h>
96
97 #include <net/pfkeyv2.h>
98 #include <inet/sadb.h>
99 #include <inet/ipsec_impl.h>
100 #include <inet/iptun/iptun_impl.h>
101 #include <inet/ipdrop.h>
102 #include <inet/ip_netinfo.h>
103 #include <inet/ilb_ip.h>
104
105 #include <sys/ethernet.h>
106 #include <net/if_types.h>
107 #include <sys/cpuvar.h>
108
109 #include <ipp/ipp.h>
110 #include <ipp/ipp_impl.h>
111 #include <ipp/ipgpc/ipgpc.h>
112
113 #include <sys/pattn.h>
114 #include <inet/dccp/dccp_ip.h>
115 #include <inet/dccp/dccp_impl.h>
116 #endif /* ! codereview */
117 #include <inet/ipclassifier.h>
118 #include <inet/sctp_ip.h>
119 #include <inet/sctp/sctp_impl.h>
120 #include <inet/udp_impl.h>
121 #include <inet/rawip_impl.h>
122 #include <inet/rts_impl.h>
123
124 #include <sys/tsol/label.h>
125 #include <sys/tsol/tnet.h>
126
127 #include <sys/queue_impl.h>
```



```

128 #include <inet/ip_arp.h>

130 #include <sys/clock_impl.h>      /* For LBOLT_FASTPATH{,64} */

132 /*
133  * Values for squeue switch:
134  * IP_SQUEUE_ENTER_NODRAIN: SQ_NODRAIN
135  * IP_SQUEUE_ENTER: SQ_PROCESS
136  * IP_SQUEUE_FILL: SQ_FILL
137  */
138 int ip_squeue_enter = IP_SQUEUE_ENTER; /* Setable in /etc/system */

140 int ip_squeue_flag;

142 /*
143  * Setable in /etc/system
144  */
145 int ip_poll_normal_ms = 100;
146 int ip_poll_normal_ticks = 0;
147 int ip_modclose_ackwait_ms = 3000;

149 /*
150  * It would be nice to have these present only in DEBUG systems, but the
151  * current design of the global symbol checking logic requires them to be
152  * unconditionally present.
153  */
154 uint_t ip_thread_data;          /* TSD key for debug support */
155 krwlock_t ip_thread_rwlock;
156 list_t ip_thread_list;

158 /*
159  * Structure to represent a linked list of msgblks. Used by ip_snmp_ functions.
160  */

162 struct listptr_s {
163     mblk_t *lp_head;          /* pointer to the head of the list */
164     mblk_t *lp_tail;        /* pointer to the tail of the list */
165 };

167 typedef struct listptr_s listptr_t;

169 /*
170  * This is used by ip_snmp_get_mib2_ip_route_media and
171  * ip_snmp_get_mib2_ip6_route_media to carry the lists of return data.
172  */
173 typedef struct iproutedata_s {
174     uint_t ird_idx;
175     uint_t ird_flags;        /* see below */
176     listptr_t ird_route;    /* ipRouteEntryTable */
177     listptr_t ird_netmedia; /* ipNetToMediaEntryTable */
178     listptr_t ird_attrs;    /* ipRouteAttributeTable */
179 } iproutedata_t;

181 /* Include ire_testhidden and IRE_IF_CLONE routes */
182 #define IRD_REPORT_ALL 0x01

184 /*
185  * Cluster specific hooks. These should be NULL when booted as a non-cluster
186  */

188 /*
189  * Hook functions to enable cluster networking
190  * On non-clustered systems these vectors must always be NULL.
191  */
192 * Hook function to Check ip specified ip address is a shared ip address
193 * in the cluster

```

```

194 *
195 */
196 int (*cl_inet_isclusterwide)(netstackid_t stack_id, uint8_t protocol,
197     sa_family_t addr_family, uint8_t *laddrp, void *args) = NULL;

199 /*
200  * Hook function to generate cluster wide ip fragment identifier
201  */
202 uint32_t (*cl_inet_ipident)(netstackid_t stack_id, uint8_t protocol,
203     sa_family_t addr_family, uint8_t *laddrp, uint8_t *faddrp,
204     void *args) = NULL;

206 /*
207  * Hook function to generate cluster wide SPI.
208  */
209 void (*cl_inet_getspi)(netstackid_t, uint8_t, uint8_t *, size_t,
210     void *) = NULL;

212 /*
213  * Hook function to verify if the SPI is already utilized.
214  */

216 int (*cl_inet_checkspi)(netstackid_t, uint8_t, uint32_t, void *) = NULL;

218 /*
219  * Hook function to delete the SPI from the cluster wide repository.
220  */

222 void (*cl_inet_deletespi)(netstackid_t, uint8_t, uint32_t, void *) = NULL;

224 /*
225  * Hook function to inform the cluster when packet received on an IDLE SA
226  */

228 void (*cl_inet_idlesa)(netstackid_t, uint8_t, uint32_t, sa_family_t,
229     in6_addr_t, in6_addr_t, void *) = NULL;

231 /*
232  * Synchronization notes:
233  *
234  * IP is a fully D_MP STREAMS module/driver. Thus it does not depend on any
235  * MT level protection given by STREAMS. IP uses a combination of its own
236  * internal serialization mechanism and standard Solaris locking techniques.
237  * The internal serialization is per phyint. This is used to serialize
238  * plumbing operations, IPMP operations, most set ioctls, etc.
239  *
240  * Plumbing is a long sequence of operations involving message
241  * exchanges between IP, ARP and device drivers. Many set ioctls are typically
242  * involved in plumbing operations. A natural model is to serialize these
243  * ioctls one per ill. For example plumbing of hme0 and qfe0 can go on in
244  * parallel without any interference. But various set ioctls on hme0 are best
245  * serialized, along with IPMP operations and processing of DLPI control
246  * messages received from drivers on a per phyint basis. This serialization is
247  * provided by the ipsq_t and primitives operating on this. Details can
248  * be found in ip_if.c above the core primitives operating on ipsq_t.
249  *
250  * Lookups of an ipif or ill by a thread return a refheld ipif / ill.
251  * Similarly lookup of an ire by a thread also returns a refheld ire.
252  * In addition ipif's and ill's referenced by the ire are also indirectly
253  * refheld. Thus no ipif or ill can vanish as long as an ipif is refheld
254  * directly or indirectly. For example an SIOCCLIFADDR ioctl that changes the
255  * address of an ipif has to go through the ipsq_t. This ensures that only
256  * one such exclusive operation proceeds at any time on the ipif. It then
257  * waits for all refcnts
258  * associated with this ipif to come down to zero. The address is changed
259  * only after the ipif has been quiesced. Then the ipif is brought up again.

```

```

260 * More details are described above the comment in ip_sioctl_flags.
261 *
262 * Packet processing is based mostly on IREs and are fully multi-threaded
263 * using standard Solaris MT techniques.
264 *
265 * There are explicit locks in IP to handle:
266 * - The ip_g_head list maintained by mi_open_link() and friends.
267 *
268 * - The reassembly data structures (one lock per hash bucket)
269 *
270 * - conn_lock is meant to protect conn_t fields. The fields actually
271 *   protected by conn_lock are documented in the conn_t definition.
272 *
273 * - ire_lock to protect some of the fields of the ire, IRE tables
274 *   (one lock per hash bucket). Refer to ip_ire.c for details.
275 *
276 * - ndp_g_lock and ncec_lock for protecting NCEs.
277 *
278 * - ill_lock protects fields of the ill and ipif. Details in ip.h
279 *
280 * - ill_g_lock: This is a global reader/writer lock. Protects the following
281 *   * The AVL tree based global multi list of all ill's.
282 *   * The linked list of all ipifs of an ill
283 *   * The <ipsq-xop> mapping
284 *   * <ill-phyint> association
285 * Insertion/deletion of an ill in the system, insertion/deletion of an ipif
286 * into an ill, changing the <ipsq-xop> mapping of an ill, changing the
287 * <ill-phyint> assoc of an ill will all have to hold the ill_g_lock as
288 * writer for the actual duration of the insertion/deletion/change.
289 *
290 * - ill_lock: This is a per ill mutex.
291 * It protects some members of the ill_t struct; see ip.h for details.
292 * It also protects the <ill-phyint> assoc.
293 * It also protects the list of ipifs hanging off the ill.
294 *
295 * - ipsq_lock: This is a per ipsq_t mutex lock.
296 * This protects some members of the ipsq_t struct; see ip.h for details.
297 * It also protects the <ipsq-ixop> mapping
298 *
299 * - ipx_lock: This is a per ipxop_t mutex lock.
300 * This protects some members of the ipxop_t struct; see ip.h for details.
301 *
302 * - phyint_lock: This is a per phyint mutex lock. Protects just the
303 *   phyint_flags
304 *
305 * - ip_addr_avail_lock: This is used to ensure the uniqueness of IP addresses.
306 * This lock is held in ipif_up_done and the ipif is marked IPIF_UP and the
307 * uniqueness check also done atomically.
308 *
309 * - ill_g_usersrc_lock: This readers/writer lock protects the usersrc
310 *   group list linked by ill_usersrc_grp_next. It also protects the
311 *   ill_usersrc_ifindex field. It is taken as a writer when a member of the
312 *   group is being added or deleted. This lock is taken as a reader when
313 *   walking the list/group(eg: to get the number of members in a usersrc group).
314 * Note, it is only necessary to take this lock if the ill_usersrc_grp_next
315 * field is changing state i.e from NULL to non-NULL or vice-versa. For
316 * example, it is not necessary to take this lock in the initial portion
317 * of ip_sioctl_slifusersrc or at all in ip_sioctl_flags since these
318 * operations are executed exclusively and that ensures that the "usersrc
319 * group state" cannot change. The "usersrc group state" change can happen
320 * only in the latter part of ip_sioctl_slifusersrc and in ill_delete.
321 *
322 * Changing <ill-phyint>, <ipsq-xop> associations:
323 *
324 * To change the <ill-phyint> association, the ill_g_lock must be held
325 * as writer, and the ill_locks of both the v4 and v6 instance of the ill

```

```

326 * must be held.
327 *
328 * To change the <ipsq-xop> association, the ill_g_lock must be held as
329 * writer, the ipsq_lock must be held, and one must be writer on the ipsq.
330 * This is only done when ill's are added or removed from IPMP groups.
331 *
332 * To add or delete an ipif from the list of ipifs hanging off the ill,
333 * ill_g_lock (writer) and ill_lock must be held and the thread must be
334 * a writer on the associated ipsq.
335 *
336 * To add or delete an ill to the system, the ill_g_lock must be held as
337 * writer and the thread must be a writer on the associated ipsq.
338 *
339 * To add or delete an ilm to an ill, the ill_lock must be held and the thread
340 * must be a writer on the associated ipsq.
341 *
342 * Lock hierarchy
343 *
344 * Some lock hierarchy scenarios are listed below.
345 *
346 * ill_g_lock -> conn_lock -> ill_lock -> ipsq_lock -> ipx_lock
347 * ill_g_lock -> ill_lock(s) -> phyint_lock
348 * ill_g_lock -> ndp_g_lock -> ill_lock -> ncec_lock
349 * ill_g_lock -> ip_addr_avail_lock
350 * conn_lock -> irb_lock -> ill_lock -> ire_lock
351 * ill_g_lock -> ip_g_nd_lock
352 * ill_g_lock -> ips_ipmp_lock -> ill_lock -> nce_lock
353 * ill_g_lock -> ndp_g_lock -> ill_lock -> ncec_lock -> nce_lock
354 * arl_lock -> ill_lock
355 * ips_ire_dep_lock -> irb_lock
356 *
357 * When more than 1 ill lock is needed to be held, all ill lock addresses
358 * are sorted on address and locked starting from highest addressed lock
359 * downward.
360 *
361 * Multicast scenarios
362 * ips_ill_g_lock -> ill_mcast_lock
363 * conn_ilg_lock -> ips_ill_g_lock -> ill_lock
364 * ill_mcast_serializer -> ill_mcast_lock -> ips_ipmp_lock -> ill_lock
365 * ill_mcast_serializer -> ill_mcast_lock -> connf_lock -> conn_lock
366 * ill_mcast_serializer -> ill_mcast_lock -> conn_ilg_lock
367 * ill_mcast_serializer -> ill_mcast_lock -> ips_igmp_timer_lock
368 *
369 * IPsec scenarios
370 *
371 * ipsa_lock -> ill_g_lock -> ill_lock
372 * ill_g_usersrc_lock -> ill_g_lock -> ill_lock
373 *
374 * Trusted Solaris scenarios
375 *
376 * igsa_lock -> gcgrp_rwlock -> gcgrp_lock
377 * igsa_lock -> gcdb_lock
378 * gcgrp_rwlock -> ire_lock
379 * gcgrp_rwlock -> gcdb_lock
380 *
381 * squeue(sq_lock), flow related (ft_lock, fe_lock) locking
382 *
383 * cpu_lock --> ill_lock --> sqset_lock --> sq_lock
384 * sq_lock -> conn_lock -> QLOCK(q)
385 * ill_lock -> ft_lock -> fe_lock
386 *
387 * Routing/forwarding table locking notes:
388 *
389 * Lock acquisition order: Radix tree lock, irb_lock.
390 * Requirements:
391 * 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_refcnt
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 happens 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 *
483 *
484 *
485 *
486 *
487 * |idl_tx_list[0]|->
488 * |-----|
489 * |-----|
490 * |-----|
491 * |-----|
492 * |-----|
493 * |-----|
494 * |-----|
495 * |-----|
496 * |-----|
497 * |-----|
498 * |-----|
499 * |idl_tx_list[1]|->
500 * |-----|
501 * |-----|
502 * |-----|
503 * |-----|
504 * |-----|
505 * |-----|
506 * |idl_tx_list[n]|-> ...
507 * |-----|
508 * |-----|
509 *
510 * When mac_tx() returns a cookie, the cookie is hashed into an index into
511 * ips_idl_tx_list[], and conn_drain_insert() is called with the idl_tx_list
512 * to insert the conn onto. conn_drain_insert() asserts flow control for the
513 * sockets via su_txq_full() (non-STREAMS) or QFULL on conn_wq (STREAMS).
514 * Further, conn_blocked is set to indicate that the conn is blocked.
515 *
516 * GLDv3 calls ill_flow_enable() when flow control is relieved. The cookie
517 * passed in the call to ill_flow_enable() identifies the blocked Tx ring and
518 * is again hashed to locate the appropriate idl_tx_list, which is then
519 * drained via conn_walk_drain(). conn_walk_drain() goes through each conn in
520 * the drain list and calls conn_drain_remove() to clear flow control (via
521 * calling su_txq_full() or clearing QFULL), and remove the conn from the
522 * drain list.
523 *
524 * Note that the drain list is not a single list but a (configurable) array of

```

```

524 * lists (8 elements by default). Synchronization between drain insertion and
525 * flow control wakeup is handled by using idl_txl->txl_lock, and only
526 * conn_drain_insert() and conn_drain_remove() manipulate the drain list.
527 *
528 * Flow control via STREAMS is used when ILL_DIRECT_CAPABLE() returns FALSE.
529 * On the send side, if the packet cannot be sent down to the driver by IP
530 * (canput() fails), ip_xmit() drops the packet and returns EWOULDBLOCK to the
531 * caller, who may then invoke ixa_check_drain_insert() to insert the conn on
532 * the 0'th drain list. When ip_wsrv() runs on the ill_wq because flow
533 * control has been relieved, the blocked conns in the 0'th drain list are
534 * drained as in the non-STREAMS case.
535 *
536 * In both the STREAMS and non-STREAMS cases, the sockfs upcall to set QFULL
537 * is done when the conn is inserted into the drain list (conn_drain_insert())
538 * and cleared when the conn is removed from the it (conn_drain_remove()).
539 *
540 * IPQoS notes:
541 *
542 * IPQoS Policies are applied to packets using IPPF (IP Policy framework)
543 * and IPQoS modules. IPPF includes hooks in IP at different control points
544 * (callout positions) which direct packets to IPQoS modules for policy
545 * processing. Policies, if present, are global.
546 *
547 * The callout positions are located in the following paths:
548 *     o local_in (packets destined for this host)
549 *     o local_out (packets originating from this host)
550 *     o fwd_in (packets forwarded by this m/c - inbound)
551 *     o fwd_out (packets forwarded by this m/c - outbound)
552 * Hooks at these callout points can be enabled/disabled using the ndd variable
553 * ip_policy_mask (a bit mask with the 4 LSB indicating the callout positions).
554 * By default all the callout positions are enabled.
555 *
556 * Outbound (local_out)
557 * Hooks are placed in ire_send_wire_v4 and ire_send_wire_v6.
558 *
559 * Inbound (local_in)
560 * Hooks are placed in ip_fanout_v4 and ip_fanout_v6.
561 *
562 * Forwarding (in and out)
563 * Hooks are placed in ire_rcv_forward_v4/v6.
564 *
565 * IP Policy Framework processing (IPPF processing)
566 * Policy processing for a packet is initiated by ip_process, which ascertains
567 * that the classifier (ipgpc) is loaded and configured, failing which the
568 * packet resumes normal processing in IP. If the classifier is present, the
569 * packet is acted upon by one or more IPQoS modules (action instances), per
570 * filters configured in ipgpc and resumes normal IP processing thereafter.
571 * An action instance can drop a packet in course of its processing.
572 *
573 * Zones notes:
574 *
575 * The partitioning rules for networking are as follows:
576 * 1) Packets coming from a zone must have a source address belonging to that
577 * zone.
578 * 2) Packets coming from a zone can only be sent on a physical interface on
579 * which the zone has an IP address.
580 * 3) Between two zones on the same machine, packet delivery is only allowed if
581 * there's a matching route for the destination and zone in the forwarding
582 * table.
583 * 4) The TCP and UDP port spaces are per-zone; that is, two processes in
584 * different zones can bind to the same port with the wildcard address
585 * (INADDR_ANY).
586 *
587 * The granularity of interface partitioning is at the logical interface level.
588 * Therefore, every zone has its own IP addresses, and incoming packets can be
589 * attributed to a zone unambiguously. A logical interface is placed into a zone

```

```

590 * using the SIOCCLIFZONE ioctl; this sets the ipif_zoneid field in the ipif_t
591 * structure. Rule (1) is implemented by modifying the source address selection
592 * algorithm so that the list of eligible addresses is filtered based on the
593 * sending process zone.
594 *
595 * The Internet Routing Entries (IREs) are either exclusive to a zone or shared
596 * across all zones, depending on their type. Here is the break-up:
597 *
598 * IRE type                               Shared/exclusive
599 * -----
600 * IRE_BROADCAST                           Exclusive
601 * IRE_DEFAULT (default routes)            Shared (*)
602 * IRE_LOCAL                               Exclusive (x)
603 * IRE_LOOPBACK                             Exclusive
604 * IRE_PREFIX (net routes)                 Shared (*)
605 * IRE_IF_NORESOLVER (interface routes)   Exclusive
606 * IRE_IF_RESOLVER (interface routes)     Exclusive
607 * IRE_IF_CLONE (interface routes)        Exclusive
608 * IRE_HOST (host routes)                  Shared (*)
609 *
610 * (*) A zone can only use a default or off-subnet route if the gateway is
611 * directly reachable from the zone, that is, if the gateway's address matches
612 * one of the zone's logical interfaces.
613 *
614 * (x) IRE_LOCAL are handled a bit differently.
615 * When ip_restrict_interzone_loopback is set (the default),
616 * ire_route_recursive restricts loopback using an IRE_LOCAL
617 * between zone to the case when L2 would have conceptually looped the packet
618 * back, i.e. the loopback which is required since neither Ethernet drivers
619 * nor Ethernet hardware loops them back. This is the case when the normal
620 * routes (ignoring IREs with different zoneids) would send out the packet on
621 * the same ill as the ill with which is IRE_LOCAL is associated.
622 *
623 * Multiple zones can share a common broadcast address; typically all zones
624 * share the 255.255.255.255 address. Incoming as well as locally originated
625 * broadcast packets must be dispatched to all the zones on the broadcast
626 * network. For directed broadcasts (e.g. 10.16.72.255) this is not trivial
627 * since some zones may not be on the 10.16.72/24 network. To handle this, each
628 * zone has its own set of IRE_BROADCAST entries; then, broadcast packets are
629 * sent to every zone that has an IRE_BROADCAST entry for the destination
630 * address on the input ill, see ip_input_broadcast().
631 *
632 * Applications in different zones can join the same multicast group address.
633 * The same logic applies for multicast as for broadcast. ip_input_multicast
634 * dispatches packets to all zones that have members on the physical interface.
635 */
637 /*
638 * Squeue Fanout flags:
639 *     0: No fanout.
640 *     1: Fanout across all squeues
641 */
642 boolean_t    ip_squeue_fanout = 0;
644 /*
645 * Maximum dups allowed per packet.
646 */
647 uint_t ip_max_frag_dups = 10;
649 static int    ip_open(queue_t *q, dev_t *devp, int flag, int sflag,
650                    cred_t *credp, boolean_t isv6);
651 static mblk_t *ip_xmit_attach_llhdr(mblk_t *, nce_t *);
653 static boolean_t icmp_inbound_verify_v4(mblk_t *, icmp_t *, ip_rcv_attr_t *);
654 static void     icmp_inbound_too_big_v4(icmp_t *, ip_rcv_attr_t *);
655 static void     icmp_inbound_error_fanout_v4(mblk_t *, icmp_t *,

```

```

656 ip_rcv_attr_t *);
657 static void icmp_options_update(ipha_t *);
658 static void icmp_param_problem(mblk_t *, uint8_t, ip_rcv_attr_t *);
659 static void icmp_pkt(mblk_t *, void *, size_t, ip_rcv_attr_t *);
660 static mblk_t *icmp_pkt_err_ok(mblk_t *, ip_rcv_attr_t *);
661 static void icmp_redirect_v4(mblk_t *mp, ipha_t *, icmp_h_t *,
662 ip_rcv_attr_t *);
663 static void icmp_send_redirect(mblk_t *, ipaddr_t, ip_rcv_attr_t *);
664 static void icmp_send_reply_v4(mblk_t *, ipha_t *, icmp_h_t *,
665 ip_rcv_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(ipseq_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 *, ipoutedata_t *);
710 static void ip_snmp_get2_v6_route(ire_t *, ipoutedata_t *);
711 static int ip_snmp_get2_v4_media(ncec_t *, ipoutedata_t *);
712 static int ip_snmp_get2_v6_media(ncec_t *, ipoutedata_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_queue_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 ioctl's 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 rtenry), IPI_PRIV,
797             MISC_CMD, ip_siocaddr, NULL },
798 /* 011 */ { SIOCDELRT, sizeof (struct rtenry), IPI_PRIV,
799             MISC_CMD, ip_siocdelrt, NULL },

801 /* 012 */ { SIOCSIFADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
802             IF_CMD, ip_siocifaddr, ip_siocifaddr_restart },
803 /* 013 */ { SIOCGIFADDR, sizeof (struct ifreq), IPI_GET_CMD,
804             IF_CMD, ip_siocifaddr_get_addr, NULL },

806 /* 014 */ { SIOCSIFDSTADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
807             IF_CMD, ip_siocifdstaddr, ip_siocifdstaddr_restart },
808 /* 015 */ { SIOCGIFDSTADDR, sizeof (struct ifreq),
809             IPI_GET_CMD, IF_CMD, ip_siocifdstaddr_get_dstaddr, NULL },

811 /* 016 */ { SIOCSIFFLAGS, sizeof (struct ifreq),
812             IPI_PRIV | IPI_WR,
813             IF_CMD, ip_siocifflags, ip_siocifflags_restart },
814 /* 017 */ { SIOCGIFFLAGS, sizeof (struct ifreq),
815             IPI_MODOK | IPI_GET_CMD,
816             IF_CMD, ip_siocifflags_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_siocifconf_get_ifconf, NULL },

825 /* 021 */ { SIOCSIFMTU, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
826             IF_CMD, ip_siocifmtu, NULL },
827 /* 022 */ { SIOCGIFMTU, sizeof (struct ifreq), IPI_GET_CMD,
828             IF_CMD, ip_siocifmtu_get_mtu, NULL },
829 /* 023 */ { SIOCGIFBRDADDR, sizeof (struct ifreq),
830             IPI_GET_CMD, IF_CMD, ip_siocifbrdaddr, NULL },
831 /* 024 */ { SIOCSIFBRDADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
832             IF_CMD, ip_siocifbrdaddr, NULL },
833 /* 025 */ { SIOCGIFNETMASK, sizeof (struct ifreq),
834             IPI_GET_CMD, IF_CMD, ip_siocifnetmask, NULL },
835 /* 026 */ { SIOCSIFNETMASK, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
836             IF_CMD, ip_siocifnetmask, ip_siocifnetmask_restart },
837 /* 027 */ { SIOCGIFMETRIC, sizeof (struct ifreq),
838             IPI_GET_CMD, IF_CMD, ip_siocifmetric, NULL },
839 /* 028 */ { SIOCSIFMETRIC, sizeof (struct ifreq), IPI_PRIV,
840             IF_CMD, ip_siocifmetric, 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_siocarp, NULL },
846 /* 031 */ { SIOCGARP, sizeof (struct arpreq), IPI_GET_CMD,
847             ARP_CMD, ip_siocarp, NULL },
848 /* 032 */ { SIOC_DARP, sizeof (struct arpreq), IPI_PRIV | IPI_WR,
849             ARP_CMD, ip_siocarp, 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_unitssel, if_unitssel_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_siocifname, 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_siocifnum_get_ifnum, NULL },
915 /* 088 */ { SIOCGIFMUXID, sizeof (struct ifreq), IPI_GET_CMD,
916             IF_CMD, ip_siocifmuxid_get_muxid, NULL },
917 /* 089 */ { SIOCSIFMUXID, sizeof (struct ifreq),
918             IPI_PRIV | IPI_WR, IF_CMD, ip_siocifmuxid, 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 */ { SIOCLIFADDR, 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 */ { SIOCLIFDSTADDR, 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 */ { SIOCLIFFLAGS, 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_SIOCGIFCONF, 0, IPI_GET_CMD, MISC_CMD,
974             ip_sioctl_get_lifconf, NULL },
975 /* 121 */ { SIOCLIFMTU, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
976             LIF_CMD, ip_sioctl_mtu, NULL },
977 /* 122 */ { SIOCGIFMTU, sizeof (struct lifreq), IPI_GET_CMD,
978             LIF_CMD, ip_sioctl_get_mtu, NULL },
979 /* 123 */ { SIOCLIFBRDADDR, 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 */ { SIOCLIFNETMASK, 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 */ { SIOCLIFMUXID, 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 */ { SIOCLIFLNKINFO, 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 */ { SIOCLIFBINDING, 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 */ { SIOCLIFGROUPNAME, 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 */ { SIOCGIP6ADDRPOLICY, 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 /* SIOCPPOCKFS is not handled by IP */
1076 /* 169 */ { IPI_DONTCARE /* SIOCPPOCKFS */, 0, 0, 0, NULL, NULL },

1078 /* 170 */ { SIOGLIFZONE, 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 */ { SIOCGPROP, 0, IPI_GET_CMD, 0, NULL, NULL },
1114 /* 189 */ { SIOCSPROP, 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     { SIOCGVIFCNT, sizeof (struct sioc_vif_req), IPI_GET_CMD,
1134          MISC_CMD, mrt_ioctl },
1135     { SIOCGTSGCNT, sizeof (struct sioc_sg_req), IPI_GET_CMD,
1136          MISC_CMD, mrt_ioctl },
1137     { SIOCGTSLSGCNT, 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 /*

```



```

1184 * Entry points for IP as a device and as a module.
1185 * We have separate open functions for the /dev/ip and /dev/ip6 devices.
1186 */
1187 static struct qinit iprinitv4 = {
1188     (pfi_t)ip_rput, NULL, ip_openv4, ip_close, NULL,
1189     &ip_mod_info
1190 };

1192 struct qinit iprinitv6 = {
1193     (pfi_t)ip_rput_v6, NULL, ip_openv6, ip_close, NULL,
1194     &ip_mod_info
1195 };

1197 static struct qinit ipwinit = {
1198     (pfi_t)ip_wput_nondata, (pfi_t)ip_wsrv, NULL, NULL, NULL,
1199     &ip_mod_info
1200 };

1202 static struct qinit iplrinit = {
1203     (pfi_t)ip_lrput, NULL, ip_openv4, ip_close, NULL,
1204     &ip_mod_info
1205 };

1207 static struct qinit iplwinit = {
1208     (pfi_t)ip_lwput, NULL, NULL, NULL, NULL,
1209     &ip_mod_info
1210 };

1212 /* For AF_INET aka /dev/ip */
1213 struct streamtab ipinfov4 = {
1214     &iprinitv4, &ipwinit, &iplrinit, &iplwinit
1215 };

1217 /* For AF_INET6 aka /dev/ip6 */
1218 struct streamtab ipinfov6 = {
1219     &iprinitv6, &ipwinit, &iplrinit, &iplwinit
1220 };

1222 #ifdef DEBUG
1223 boolean_t skip_sctp_cksum = B_FALSE;
1224 #endif

1226 /*
1227  * Generate an ICMP fragmentation needed message.
1228  * When called from ip_output side a minimal ip_rcv_attr_t needs to be
1229  * constructed by the caller.
1230  */
1231 void
1232 icmp_frag_needed(mblk_t *mp, int mtu, ip_rcv_attr_t *ira)
1233 {
1234     icmp_h_t icmp_h;
1235     ip_stack_t *ipst = ira->ira_ill->ill_ipst;

1237     mp = icmp_pkt_err_ok(mp, ira);
1238     if (mp == NULL)
1239         return;

1241     bzero(&icmp_h, sizeof(icmp_h_t));
1242     icmp_h.icmph_type = ICMP_DEST_UNREACHABLE;
1243     icmp_h.icmph_code = ICMP_FRAGMENTATION_NEEDED;
1244     icmp_h.icmph_du_mtu = htons((uint16_t)mtu);
1245     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutFragNeeded);
1246     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDestUnreachs);

1248     icmp_pkt(mp, &icmp_h, sizeof(icmp_h_t), ira);
1249 }

```

```

1251 /*
1252  * icmp_inbound_v4 deals with ICMP messages that are handled by IP.
1253  * If the ICMP message is consumed by IP, i.e., it should not be delivered
1254  * to any IPPROTO_ICMP raw sockets, then it returns NULL.
1255  * Likewise, if the ICMP error is malformed (too short, etc), then it
1256  * returns NULL. The caller uses this to determine whether or not to send
1257  * to raw sockets.
1258  *
1259  * All error messages are passed to the matching transport stream.
1260  *
1261  * The following cases are handled by icmp_inbound:
1262  * 1) It needs to send a reply back and possibly delivering it
1263  *    to the "interested" upper clients.
1264  * 2) Return the mblk so that the caller can pass it to the RAW socket clients.
1265  * 3) It needs to change some values in IP only.
1266  * 4) It needs to change some values in IP and upper layers e.g TCP
1267  *    by delivering an error to the upper layers.
1268  *
1269  * We handle the above three cases in the context of IPsec in the
1270  * following way :
1271  *
1272  * 1) Send the reply back in the same way as the request came in.
1273  *    If it came in encrypted, it goes out encrypted. If it came in
1274  *    clear, it goes out in clear. Thus, this will prevent chosen
1275  *    plain text attack.
1276  * 2) The client may or may not expect things to come in secure.
1277  *    If it comes in secure, the policy constraints are checked
1278  *    before delivering it to the upper layers. If it comes in
1279  *    clear, ipsec_inbound_accept_clear will decide whether to
1280  *    accept this in clear or not. In both the cases, if the returned
1281  *    message (IP header + 8 bytes) that caused the icmp message has
1282  *    AH/ESP headers, it is sent up to AH/ESP for validation before
1283  *    sending up. If there are only 8 bytes of returned message, then
1284  *    upper client will not be notified.
1285  * 3) Check with global policy to see whether it matches the constraints.
1286  *    But this will be done only if icmp_accept_messages_in_clear is
1287  *    zero.
1288  * 4) If we need to change both in IP and ULP, then the decision taken
1289  *    while affecting the values in IP and while delivering up to TCP
1290  *    should be the same.
1291  *
1292  *     There are two cases.
1293  *
1294  *     a) If we reject data at the IP layer (ipsec_check_global_policy()
1295  *        failed), we will not deliver it to the ULP, even though they
1296  *        are *willing* to accept in *clear*. This is fine as our global
1297  *        disposition to icmp messages asks us reject the datagram.
1298  *
1299  *     b) If we accept data at the IP layer (ipsec_check_global_policy()
1300  *        succeeded or icmp_accept_messages_in_clear is 1), and not able
1301  *        to deliver it to ULP (policy failed), it can lead to
1302  *        consistency problems. The cases known at this time are
1303  *        ICMP_DESTINATION_UNREACHABLE messages with following code
1304  *        values :
1305  *
1306  *     - ICMP_FRAGMENTATION_NEEDED : IP adapts to the new value
1307  *        and Upper layer rejects. Then the communication will
1308  *        come to a stop. This is solved by making similar decisions
1309  *        at both levels. Currently, when we are unable to deliver
1310  *        to the Upper Layer (due to policy failures) while IP has
1311  *        adjusted dce_pmtu, the next outbound datagram would
1312  *        generate a local ICMP_FRAGMENTATION_NEEDED message - which
1313  *        will be with the right level of protection. Thus the right
1314  *        value will be communicated even if we are not able to
1315  *        communicate when we get from the wire initially. But this

```

```

1316 *          assumes there would be at least one outbound datagram after
1317 *          IP has adjusted its dce_pmtu value. To make things
1318 *          simpler, we accept in clear after the validation of
1319 *          AH/ESP headers.
1320 *
1321 *          - Other ICMP ERRORS : We may not be able to deliver it to the
1322 *          upper layer depending on the level of protection the upper
1323 *          layer expects and the disposition in ipsec_inbound_accept_clear().
1324 *          ipsec_inbound_accept_clear() decides whether a given ICMP error
1325 *          should be accepted in clear when the Upper layer expects secure.
1326 *          Thus the communication may get aborted by some bad ICMP
1327 *          packets.
1328 */
1329 mblk_t *
1330 icmp_inbound_v4(mblk_t *mp, ip_rcv_attr_t *ira)
1331 {
1332     icmp_h_t      *icmph;
1333     ipha_t        *ipha;
1334     int           ip_hdr_length; /* Outer header length */
1335     boolean_t     interested;
1336     ipif_t        *ipif;
1337     uint32_t      ts;
1338     uint32_t      *tsp;
1339     timestruc_t   now;
1340     ill_t         *ill = ira->ira_ill;
1341     ip_stack_t    *ipst = ill->ill_ipst;
1342     zoneid_t      zoneid = ira->ira_zoneid;
1343     int           len_needed;
1344     mblk_t        *mp_ret = NULL;
1345
1346     ipha = (ipha_t *)mp->b_rptr;
1347
1348     BUMP_MIB(&ipst->ips_icmp_mib, icmpInMsgs);
1349
1350     ip_hdr_length = ira->ira_ip_hdr_length;
1351     if ((mp->b_wptr - mp->b_rptr) < (ip_hdr_length + ICMPH_SIZE)) {
1352         if (ira->ira_pktlen < (ip_hdr_length + ICMPH_SIZE)) {
1353             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
1354             ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
1355             freemsg(mp);
1356             return (NULL);
1357         }
1358         /* Last chance to get real. */
1359         ipha = ip_pullup(mp, ip_hdr_length + ICMPH_SIZE, ira);
1360         if (ipha == NULL) {
1361             BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
1362             freemsg(mp);
1363             return (NULL);
1364         }
1365     }
1366
1367     /* The IP header will always be a multiple of four bytes */
1368     icmph = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1369     ip2dbg(("icmp_inbound_v4: type %d code %d\n", icmph->icmp_type,
1370           icmph->icmp_code));
1371
1372     /*
1373      * We will set "interested" to "true" if we should pass a copy to
1374      * the transport or if we handle the packet locally.
1375      */
1376     interested = B_FALSE;
1377     switch (icmph->icmp_type) {
1378     case ICMP_ECHO_REPLY:
1379         BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchoReps);
1380         break;
1381     case ICMP_DEST_UNREACHABLE:

```

```

1382         if (icmph->icmp_code == ICMP_FRAGMENTATION_NEEDED)
1383             BUMP_MIB(&ipst->ips_icmp_mib, icmpInFragNeeded);
1384         interested = B_TRUE; /* Pass up to transport */
1385         BUMP_MIB(&ipst->ips_icmp_mib, icmpInDestUnreachs);
1386         break;
1387     case ICMP_SOURCE_QUENCH:
1388         interested = B_TRUE; /* Pass up to transport */
1389         BUMP_MIB(&ipst->ips_icmp_mib, icmpInSrcQuenchs);
1390         break;
1391     case ICMP_REDIRECT:
1392         if (!ipst->ips_ip_ignore_redirect)
1393             interested = B_TRUE;
1394         BUMP_MIB(&ipst->ips_icmp_mib, icmpInRedirects);
1395         break;
1396     case ICMP_ECHO_REQUEST:
1397         /*
1398          * Whether to respond to echo requests that come in as IP
1399          * broadcasts or as IP multicast is subject to debate
1400          * (what isn't?). We aim to please, you pick it.
1401          * Default is do it.
1402          */
1403         if (ira->ira_flags & IRAF_MULTICAST) {
1404             /* multicast: respond based on tunable */
1405             interested = ipst->ips_ip_g_resp_to_echo_mcast;
1406         } else if (ira->ira_flags & IRAF_BROADCAST) {
1407             /* broadcast: respond based on tunable */
1408             interested = ipst->ips_ip_g_resp_to_echo_bcst;
1409         } else {
1410             /* unicast: always respond */
1411             interested = B_TRUE;
1412         }
1413         BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchos);
1414         if (!interested) {
1415             /* We never pass these to RAW sockets */
1416             freemsg(mp);
1417             return (NULL);
1418         }
1419
1420         /* Check db_ref to make sure we can modify the packet. */
1421         if (mp->b_datap->db_ref > 1) {
1422             mblk_t *mpl;
1423
1424             mpl = copymsg(mp);
1425             freemsg(mp);
1426             if (!mpl) {
1427                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1428                 return (NULL);
1429             }
1430             mp = mpl;
1431             ipha = (ipha_t *)mp->b_rptr;
1432             icmph = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1433         }
1434         icmph->icmp_type = ICMP_ECHO_REPLY;
1435         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutEchoReps);
1436         icmp_send_reply_v4(mp, ipha, icmph, ira);
1437         return (NULL);
1438
1439     case ICMP_ROUTER_ADVERTISEMENT:
1440     case ICMP_ROUTER_SOLICITATION:
1441         break;
1442     case ICMP_TIME_EXCEEDED:
1443         interested = B_TRUE; /* Pass up to transport */
1444         BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimeExcds);
1445         break;
1446     case ICMP_PARAM_PROBLEM:
1447         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     }
1464
1465     /* Make sure we have enough of the packet */
1466     len_needed = ip_hdr_length + ICMPH_SIZE +
1467                 3 * sizeof (uint32_t);
1468
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         icmp_h = (icmp_h_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 *mpl;
1485
1486         mpl = copymsg(mp);
1487         freemsg(mp);
1488         if (!mpl) {
1489             BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1490             return (NULL);
1491         }
1492         mp = mpl;
1493         ipha = (ipha_t *)mp->b_rptr;
1494         icmp_h = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1495     }
1496     icmp_h->icmp_h_type = ICMP_TIME_STAMP_REPLY;
1497     tsp = (uint32_t *)&icmp_h[1];
1498     tsp++; /* Skip past 'originate time' */
1499     /* Compute # of milliseconds since midnight */
1500     gethrtime(&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, icmp_h, ira);
1507     return (NULL);
1508
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_respnd_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         icmp_h = (icmp_h_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 *mpl;
1546
1547         mpl = copymsg(mp);
1548         freemsg(mp);
1549         if (!mpl) {
1550             BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1551             return (NULL);
1552         }
1553         mp = mpl;
1554         ipha = (ipha_t *)mp->b_rptr;
1555         icmp_h = (icmp_h_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     icmp_h->icmp_h_type = ICMP_ADDRESS_MASK_REPLY;
1573     bcopy(&ipif->ipif_net_mask, &icmp_h[1], IP_ADDR_LEN);
1574     ipif_refrele(ipif);
1575     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutAddrMaskReps);
1576     icmp_send_reply_v4(mp, ipha, icmp_h, ira);
1577     return (NULL);
1578
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 *mpl;

1627         mpl = copymsg(mp);
1628         if (mpl == 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 = mpl;
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_rcv_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.ixaf_flags = IXAF_BASIC_SIMPLE_V4;
1700     ixas.ixaz_zoneid = ira->ira_zoneid;
1701     ixas.ixac_cred = kcred;
1702     ixas.ixacpid = NOPID;
1703     ixas.ixat_sl = ira->ira_sl;    /* Behave as a multi-level responder */
1704     ixas.ixai_ifindex = 0;
1705     ixas.ixai_ipst = ipst;
1706     ixas.ixam_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.ixaf_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 of our addresses (IRE_LOCALS), thus we let
1725          * ip_output_simple pick the source.
1726          */
1727         ipha->ipha_src = INADDR_ANY;
1728         ixas.ixaf_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.ixaf_flags |= IXAF_PMTU_DISCOVERY;
1733         ipha->ipha_fragment_offset_and_flags |= IPH_DF_HTONS;
1734     }
1735
1736     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutMsgs);
1737
1738     (void) ip_output_simple(mp, &ixas);
1739     ixa_cleanup(&ixas);
1740 }
1741
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  * Called before icmp_inbound_error_fanout_v4 is called.
1748  */
1749
1750 static boolean_t
1751 icmp_inbound_verify_v4(mblk_t *mp, icmph_t *icmph, ip_rcv_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 */
1758
1759     ipha = (ipha_t *)&icmph[1];
1760     if ((uchar_t *)ipha + IP_SIMPLE_HDR_LENGTH > mp->b_wptr)
1761         goto truncated;
1762
1763     hdr_length = IPH_HDR_LENGTH(ipha);
1764
1765     if ((IPH_HDR_VERSION(ipha) != IPV4_VERSION))
1766         goto discard_pkt;
1767
1768     if (hdr_length < sizeof (ipha_t))
1769         goto truncated;
1770
1771     if ((uchar_t *)ipha + hdr_length > mp->b_wptr)
1772         goto truncated;
1773
1774     /*
1775      * Stop here for ICMP_REDIRECT.
1776      */
1777     if (icmph->icmph_type == ICMP_REDIRECT)

```

```

1778         return (B_TRUE);
1779
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;
1795
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;
1803
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;
1809
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     }
1838
1839     return (B_TRUE);
1840
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 everything 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(icmp_t *icmp, ip_rcv_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 *)&icmp[1];
1887     ASSERT(icmp->icmp_type == ICMP_DEST_UNREACHABLE &&
1888         icmp->icmp_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         ipldbg("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(icmp->icmp_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 (icmp->icmp_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             ipldbg("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             ipldbg("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             ipldbg("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     icmp->icmp_du_zero = 0;
1969     icmp->icmp_du_mtu = htons((uint16_t)dce->dce_pmtu);
1970     DTRACE_PROBE4(ip4_pmtu_change, icmp_t *, icmp, 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 *
2043 * Note that we retain ixa_fragsize across IPsec thus once
2044 * we have picking ixa_fragsize and entered ipsec_out_process we do
2045 * no change the fragsize even if the path MTU changes before
2046 * we reach ip_output_post_ipsec.
2047 *
2048 * In the local case, IRAF_LOOPBACK will be set indicating
2049 * that IFN was generated locally.
2050 *
2051 * ROUTER : IFN could be secure or non-secure.
2052 *
2053 * SECURE : We use the IPSEC_IN to fanout to AH/ESP if the
2054 * packet in error has AH/ESP headers to validate the AH/ESP
2055 * headers. AH/ESP will verify whether there is a valid SA or
2056 * not and send it back. We will fanout again if we have more
2057 * data in the packet.
2058 *
2059 * If the packet in error does not have AH/ESP, we handle it
2060 * like any other case.
2061 *
2062 * NON_SECURE : If the packet in error has AH/ESP headers, we send it
2063 * up to AH/ESP for validation. AH/ESP will verify whether there is a
2064 * valid SA or not and send it back. We will fanout again if
2065 * we have more data in the packet.
2066 *
2067 * If the packet in error does not have AH/ESP, we handle it
2068 * like any other case.
2069 *
2070 * The caller must have called icmp_inbound_verify_v4.
2071 */
2072 static void
2073 icmp_inbound_error_fanout_v4(mblk_t *mp, icmp_h_t *icmp_h, ip_rcv_attr_t *ira)
2074 {
2075     uint16_t *up; /* Pointer to ports in ULP header */
2076     uint32_t ports; /* reversed ports for fanout */
2077     ipha_t riph; /* With reversed addresses */
2078     ipha_t *ipha; /* Inner IP header */
2079     uint_t hdr_length; /* Inner IP header length */
2080     tcp_h_t *tcp_h;
2081     conn_t *conn;
2082     ill_t *ill = ira->ira_ill;
2083     ip_stack_t *ipst = ill->ill_ipst;
2084     ipsec_stack_t *ipss = ipst->ips_netstack->netstack_ipsec;
2085     ill_t *rill = ira->ira_rill;

2086     /* Caller already pulled up everything. */
2087     ipha = (ipha_t *)&icmp_h[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     riph.ipha_src = ipha->ipha_dst;
2100     riph.ipha_dst = ipha->ipha_src;
2101     riph.ipha_protocol = ipha->ipha_protocol;
2102     riph.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            riph.ipha_protocol, ntohl(iph->ipha_src),
2106            ntohl(iph->ipha_dst),
2107            icmp_h->icmp_type, icmp_h->icmp_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         tcpa = (tcpa_t *)((uchar_t *)ipha + hdr_length);
2130         connp = ipcl_tcp_lookup_reversed_ipv4(ipha, tcpa, 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_sq, 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_rcv)(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-encapsulated.
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     ipldb("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 everything 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 totalen; /* total length of all options */
2334     totalen = ipha->ipha_version_and_hdr_length -
2335     (uint8_t)((IP_VERSION << 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);
2336     totalen <= 2;
2337     optp->ipoptp_next = (uint8_t *)&ipha[1];
2338     optp->ipoptp_end = optp->ipoptp_next + totalen;
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 totalen, uint8_t *opt)
2346 {
2347     optp->ipoptp_next = opt;
2348     optp->ipoptp_end = optp->ipoptp_next + totalen;
2349     optp->ipoptp_flags = 0;
2350     return (ipoptp_next(opt));
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 implicitly 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    optp;
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    totalen;
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     totalen = ipp->ipp_ipv4_options_len;
2487     if (totalen & 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(&optp, totalen, ipp->ipp_ipv4_options);
2497          optval != IPOPT_EOL;
2498          optval = ipoptp_next(&optp)) {
2499         int    off;
2500
2501         opt = optp.ipoptp_cur;
2502         if ((optp.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(("icmp_options_update: bad src route\n"));
2600                 break;
2601             }
2602             bcopy((char *)opt + off2, &dst, IP_ADDR_LEN);
2603             bcopy(&ipha->ipha_dst, (char *)opt + off2, IP_ADDR_LEN);
2604             bcopy(&dst, &ipha->ipha_dst, IP_ADDR_LEN);
2605             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_fhtable_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_fhtable_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 }
2671
2672 ire_refrele(prev_ire);
2673 ire_refrele(ire);
2674
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: /* TODO: TOS specificity for cases 2 and 3 */
2682 case 2:
2683 case 3:
2684     break;
2685 default:
2686     BUMP_MIB(&ipst->ips_icmp_mib, icmpInBadRedirects);
2687     ip_drop_input("icmpInBadRedirects - code", mp, ira->ira_ill);
2688     freemsg(mp);
2689     return;
2690 }
2691 /*
2692  * Create a Route Association. This will allow us to remember that
2693  * someone we believe told us to use the particular gateway.
2694  */
2695 ire = ire_create(
2696     (uchar_t *)&dst, /* dest addr */
2697     (uchar_t *)&ip_g_all_ones, /* mask */
2698     (uchar_t *)&gateway, /* gateway addr */
2699     IRE_HOST,
2700     NULL, /* ill */

```

```

2702     ALL_ZONES,
2703     (RTF_DYNAMIC | RTF_GATEWAY | RTF_HOST),
2704     NULL, /* tsol_gc_t */
2705     ipst);
2706
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 */
2722 }
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 }
2728 /*
2729  * Delete any existing IRE_HOST type redirect ireds for this destination.
2730  * This together with the added IRE has the effect of
2731  * modifying an existing redirect.
2732  */
2733 prev_ire = ire_fhtable_lookup_v4(dst, 0, src, IRE_HOST, NULL,
2734     ALL_ZONES, NULL, (MATCH_IRE_GW | MATCH_IRE_TYPE), 0, ipst, NULL);
2735 if (prev_ire != NULL) {
2736     if (prev_ire->ire_flags & RTF_DYNAMIC)
2737         ire_delete(prev_ire);
2738     ire_refrele(prev_ire);
2739 }
2740
2741 freemsg(mp);
2742 }
2743
2744 /*
2745  * Generate an ICMP parameter problem message.
2746  * When called from ip_output side a minimal ip_recv_attr_t needs to be
2747  * constructed by the caller.
2748  */
2749 static void
2750 icmp_param_problem(mblk_t *mp, uint8_t ptr, ip_recv_attr_t *ira)
2751 {
2752     icmph_t icmph;
2753     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
2754
2755     mp = icmp_pkt_err_ok(mp, ira);
2756     if (mp == NULL)
2757         return;
2758
2759     bzero(&icmph, sizeof (icmph_t));
2760     icmph.icmph_type = ICMP_PARAM_PROBLEM;
2761     icmph.icmph_pp_ptr = ptr;
2762     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutParmProbs);
2763     icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
2764 }
2765
2766 /*

```

```

2768 * Build and ship an IPv4 ICMP message using the packet data in mp, and
2769 * the ICMP header pointed to by "stuff". (May be called as writer.)
2770 * Note: assumes that icmp_pkt_err_ok has been called to verify that
2771 * an icmp error packet can be sent.
2772 * Assigns an appropriate source address to the packet. If ipha_dst is
2773 * one of our addresses use it for source. Otherwise let ip_output_simple
2774 * pick the source address.
2775 */
2776 static void
2777 icmp_pkt(mblk_t *mp, void *stuff, size_t len, ip_rcv_attr_t *ira)
2778 {
2779     ipaddr_t dst;
2780     icmp_h_t *icmp_h;
2781     ipha_t *ipha;
2782     uint_t len_needed;
2783     size_t msg_len;
2784     mblk_t *mpl;
2785     ipaddr_t src;
2786     ire_t *ire;
2787     ip_xmit_attr_t ixas;
2788     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
2790     ipha = (ipha_t *)mp->b_rptr;
2792     bzero(&ixas, sizeof(ixas));
2793     ixas.ixas_ixas_ixaf_flags = IXAF_BASIC_SIMPLE_V4;
2794     ixas.ixas_ixas_ixaf_zoneid = ira->ira_zoneid;
2795     ixas.ixas_ixas_ixaf_ifindex = 0;
2796     ixas.ixas_ixas_ixaf_ipst = ipst;
2797     ixas.ixas_ixas_ixaf_cred = kcred;
2798     ixas.ixas_ixas_ixaf_cpuid = NOPID;
2799     ixas.ixas_ixas_ixaf_tsl = ira->ira_tsl; /* Behave as a multi-level responder */
2800     ixas.ixas_ixas_ixaf_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;
2802     if (ira->ira_ixaf_flags & IRAF_IPSEC_SECURE) {
2803         /*
2804          * Apply IPsec based on how IPsec was applied to
2805          * the packet that had the error.
2806          *
2807          * If it was an outbound packet that caused the ICMP
2808          * error, then the caller will have setup the IRA
2809          * appropriately.
2810          */
2811         if (!ipsec_in_to_out(ira, &ixas, mp, ipha, NULL)) {
2812             BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsOutDiscards);
2813             /* Note: mp already consumed and ip_drop_packet done */
2814             return;
2815         }
2816     } else {
2817         /*
2818          * This is in clear. The icmp message we are building
2819          * here should go out in clear, independent of our policy.
2820          */
2821         ixas.ixas_ixas_ixaf_flags |= IXAF_NO_IPSEC;
2822     }
2824     /* Remember our eventual destination */
2825     dst = ipha->ipha_src;
2827     /*
2828     * If the packet was for one of our unicast addresses, make
2829     * sure we respond with that as the source. Otherwise
2830     * have ip_output_simple pick the source address.
2831     */
2832     ire = ire_fhtable_lookup_v4(ipha->ipha_dst, 0, 0,
2833         (IRE_LOCAL|IRE_LOOPBACK), NULL, ira->ira_zoneid, NULL,

```

```

2834     MATCH_IRE_TYPE|MATCH_IRE_ZONEONLY, 0, ipst, NULL);
2835     if (ire != NULL) {
2836         ire_refrele(ire);
2837         src = ipha->ipha_dst;
2838     } else {
2839         src = INADDR_ANY;
2840         ixas.ixas_ixas_ixaf_flags |= IXAF_SET_SOURCE;
2841     }
2843     /*
2844     * Check if we can send back more then 8 bytes in addition to
2845     * the IP header. We try to send 64 bytes of data and the internal
2846     * header in the special cases of ipv4 encapsulated ipv4 or ipv6.
2847     */
2848     len_needed = IPH_HDR_LENGTH(ipha);
2849     if (ipha->ipha_ixaf_protocol == IPPROTO_ENCAP ||
2850         ipha->ipha_ixaf_protocol == IPPROTO_IPV6) {
2851         if (!pullupmsg(mp, -1)) {
2852             BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsOutDiscards);
2853             ip_drop_output("ipIfStatsOutDiscards", mp, NULL);
2854             freemsg(mp);
2855             return;
2856         }
2857         ipha = (ipha_t *)mp->b_rptr;
2859         if (ipha->ipha_ixaf_protocol == IPPROTO_ENCAP) {
2860             len_needed += IPH_HDR_LENGTH(((uchar_t *)ipha +
2861                 len_needed));
2862         } else {
2863             ip6_t *ip6h = (ip6_t *)((uchar_t *)ipha + len_needed);
2865             ASSERT(ipha->ipha_ixaf_protocol == IPPROTO_IPV6);
2866             len_needed += ip_hdr_length_v6(mp, ip6h);
2867         }
2868     }
2869     len_needed += ipst->ips_ip_icmp_return;
2870     msg_len = msgdsize(mp);
2871     if (msg_len > len_needed) {
2872         (void) adjmsg(mp, len_needed - msg_len);
2873         msg_len = len_needed;
2874     }
2875     mpl = allocb(sizeof(icmp_ipha) + len, BPRI_MED);
2876     if (mpl == NULL) {
2877         BUMP_MIB(&ipst->ips_ip_mib, icmpOutErrors);
2878         freemsg(mp);
2879         return;
2880     }
2881     mpl->b_cont = mp;
2882     mp = mpl;
2884     /*
2885     * Set IXAF_TRUSTED_ICMP so we can let the ICMP messages this
2886     * node generates be accepted in peace by all on-host destinations.
2887     * If we do NOT assume that all on-host destinations trust
2888     * self-generated ICMP messages, then rework here, ip6.c, and spd.c.
2889     * (Look for IXAF_TRUSTED_ICMP).
2890     */
2891     ixas.ixas_ixas_ixaf_flags |= IXAF_TRUSTED_ICMP;
2893     ipha = (ipha_t *)mp->b_rptr;
2894     mpl->b_wptr = (uchar_t *)ipha + (sizeof(icmp_ipha) + len);
2895     *ipha = icmp_ipha;
2896     ipha->ipha_ixaf_src = src;
2897     ipha->ipha_ixaf_dst = dst;
2898     ipha->ipha_ixaf_ttl = ipst->ips_ip_def_ttl;
2899     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     icmp_h = (icmp_h_t *)&ipha[1];
2906     bcopy(stuff, icmp_h, len);
2907     icmp_h->icmp_checksum = 0;
2908     icmp_h->icmp_checksum = IP_CSUM(mp, (int32_t)sizeof(ip_h_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 arbitrarily 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 lose 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         ipldb("icmp_err_rate_limit: %d sent in burst\n",
2962             ipst->ips_icmp_pkt_err_sent);
2963         return (B_FALSE);
2964     }
2965     ipldb("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     icmp_h_t *icmp_h;
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         icmp_h = (icmp_h_t *)
3017             (&((char *)ipha)[IPH_HDR_LENGTH(ipha)]);
3018         switch (icmp_h->icmp_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_rcv_attr_t *ira)
3060 {
3061     ip_stack_t    *ipst = ira->ira_ill->ill_ipst;
3062     ipaddr_t      src, nhop;
3063     mblk_t        *mpl;
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         mpl = copymsg(mp);
3122         if (mpl != NULL) {
3123             icmp_send_redirect(mpl, 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_rcv_attr_t *ira)
3134 {
3135     icmp_h_t icmp_h;
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(&icmp_h, sizeof (icmp_h_t));
3143     icmp_h.icmph_type = ICMP_REDIRECT;
3144     icmp_h.icmph_code = 1;
3145     icmp_h.icmph_rd_gateway = gateway;
3146     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutRedirects);
3147     icmp_pkt(mp, &icmp_h, sizeof (icmp_h_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_rcv_attr_t *ira)
3155 {
3156     icmp_h_t icmp_h;
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(&icmp_h, sizeof (icmp_h_t));

```

```

3164     icmp_h.icmph_type = ICMP_TIME_EXCEEDED;
3165     icmp_h.icmph_code = code;
3166     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutTimeExcds);
3167     icmp_pkt(mp, &icmp_h, sizeof (icmp_h_t), ira);
3168 }

3170 /*
3171  * Generate an ICMP unreachable message.
3172  * When called from ip_output side a minimal ip_rcv_attr_t needs to be
3173  * constructed by the caller.
3174  */
3175 void
3176 icmp_unreachable(mblk_t *mp, uint8_t code, ip_rcv_attr_t *ira)
3177 {
3178     icmp_h_t icmp_h;
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(&icmp_h, sizeof (icmp_h_t));
3186     icmp_h.icmph_type = ICMP_DEST_UNREACHABLE;
3187     icmp_h.icmph_code = code;
3188     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDestUnreachs);
3189     icmp_pkt(mp, &icmp_h, sizeof (icmp_h_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_rcv_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_rcv_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 = ip_latch_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 = ipc1_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;
3371     zoneid_t          zoneid = ixa->ixa_zoneid;
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 uninitialized 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);
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     }
```

```

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_output 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 #ifndef DEBUG
3465      ire_rehold_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 #ifndef DEBUG
3490      dce_rehold_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          /*

```



```

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  * Furthermore, 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 too 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->ncc_common->ncec_ill->ill_mc_mtu < pmtu)
3810     pmtu = nce->ncc_common->ncec_ill->ill_mc_mtu;
3811     if (nce->ncc_common->ncec_ill != nce->ncc_ill &&
3812     nce->ncc_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->ncc_ill->ill_mc_mtu;
3820     }
3821     } else {
3822     if (nce->ncc_common->ncec_ill->ill_mtu < pmtu)
3823     pmtu = nce->ncc_common->ncec_ill->ill_mtu;

```



```

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 enable 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     ipldb("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_mrout)
```

```

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  * Wrapper 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_rcv_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_rcv_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);

```



```

4352 #endif
4354 /*
4355  * Perform cleanup for special interfaces (loopback and IPMP).
4356  */
4357 ip_interface_cleanup(ipst);
4359 /*
4360  * The *_hook_shutdown()s start the process of notifying any
4361  * consumers that things are going away.... nothing is destroyed.
4362  */
4363 ipv4_hook_shutdown(ipst);
4364 ipv6_hook_shutdown(ipst);
4365 arp_hook_shutdown(ipst);
4367 mutex_enter(&ipst->ips_capab_taskq_lock);
4368 ipst->ips_capab_taskq_quit = B_TRUE;
4369 cv_signal(&ipst->ips_capab_taskq_cv);
4370 mutex_exit(&ipst->ips_capab_taskq_lock);
4371 }
4373 /*
4374  * Free the IP stack instance.
4375  */
4376 static void
4377 ip_stack_fini(netstackid_t stackid, void *arg)
4378 {
4379     ip_stack_t *ipst = (ip_stack_t *)arg;
4380     int ret;
4382 #ifdef NS_DEBUG
4383     printf("ip_stack_fini(%p, stack %d)\n", (void *)ipst, stackid);
4384 #endif
4385     /*
4386      * At this point, all of the notifications that the events and
4387      * protocols are going away have been run, meaning that we can
4388      * now set about starting to clean things up.
4389      */
4390     ipobs_fini(ipst);
4391     ipv4_hook_destroy(ipst);
4392     ipv6_hook_destroy(ipst);
4393     arp_hook_destroy(ipst);
4394     ip_net_destroy(ipst);
4396     ipmp_destroy(ipst);
4398     ip_kstat_fini(stackid, ipst->ips_ip_mibkp);
4399     ipst->ips_ip_mibkp = NULL;
4400     icmp_kstat_fini(stackid, ipst->ips_icmp_mibkp);
4401     ipst->ips_icmp_mibkp = NULL;
4402     ip_kstat2_fini(stackid, ipst->ips_ip_kstat);
4403     ipst->ips_ip_kstat = NULL;
4404     bzero(&ipst->ips_ip_statistics, sizeof (ipst->ips_ip_statistics));
4405     ip6_kstat_fini(stackid, ipst->ips_ip6_kstat);
4406     ipst->ips_ip6_kstat = NULL;
4407     bzero(&ipst->ips_ip6_statistics, sizeof (ipst->ips_ip6_statistics));
4409     kmem_free(ipst->ips_propinfo_tbl,
4410              ip_propinfo_count * sizeof (mod_prop_info_t));
4411     ipst->ips_propinfo_tbl = NULL;
4413     dce_stack_destroy(ipst);
4414     ip_mrouter_stack_destroy(ipst);
4416     ret = untimeout(ipst->ips_igmp_timeout_id);
4417     if (ret == -1) {

```

```

4418         ASSERT(ipst->ips_igmp_timeout_id == 0);
4419     } else {
4420         ASSERT(ipst->ips_igmp_timeout_id != 0);
4421         ipst->ips_igmp_timeout_id = 0;
4422     }
4423     ret = untimeout(ipst->ips_igmp_slowtimeout_id);
4424     if (ret == -1) {
4425         ASSERT(ipst->ips_igmp_slowtimeout_id == 0);
4426     } else {
4427         ASSERT(ipst->ips_igmp_slowtimeout_id != 0);
4428         ipst->ips_igmp_slowtimeout_id = 0;
4429     }
4430     ret = untimeout(ipst->ips_mld_timeout_id);
4431     if (ret == -1) {
4432         ASSERT(ipst->ips_mld_timeout_id == 0);
4433     } else {
4434         ASSERT(ipst->ips_mld_timeout_id != 0);
4435         ipst->ips_mld_timeout_id = 0;
4436     }
4437     ret = untimeout(ipst->ips_mld_slowtimeout_id);
4438     if (ret == -1) {
4439         ASSERT(ipst->ips_mld_slowtimeout_id == 0);
4440     } else {
4441         ASSERT(ipst->ips_mld_slowtimeout_id != 0);
4442         ipst->ips_mld_slowtimeout_id = 0;
4443     }
4445     ip_ire_fini(ipst);
4446     ip6_asp_free(ipst);
4447     conn_drain_fini(ipst);
4448     ipcl_destroy(ipst);
4450     mutex_destroy(&ipst->ips_ndp4->ndp_g_lock);
4451     mutex_destroy(&ipst->ips_ndp6->ndp_g_lock);
4452     kmem_free(ipst->ips_ndp4, sizeof (ndp_g_t));
4453     ipst->ips_ndp4 = NULL;
4454     kmem_free(ipst->ips_ndp6, sizeof (ndp_g_t));
4455     ipst->ips_ndp6 = NULL;
4457     if (ipst->ips_loopback_ksp != NULL) {
4458         kstat_delete_netstack(ipst->ips_loopback_ksp, stackid);
4459         ipst->ips_loopback_ksp = NULL;
4460     }
4462     mutex_destroy(&ipst->ips_capab_taskq_lock);
4463     cv_destroy(&ipst->ips_capab_taskq_cv);
4465     rw_destroy(&ipst->ips_srcid_lock);
4467     mutex_destroy(&ipst->ips_ip_mi_lock);
4468     rw_destroy(&ipst->ips_ill_g_usesrc_lock);
4470     mutex_destroy(&ipst->ips_igmp_timer_lock);
4471     mutex_destroy(&ipst->ips_mld_timer_lock);
4472     mutex_destroy(&ipst->ips_igmp_slowtimeout_lock);
4473     mutex_destroy(&ipst->ips_mld_slowtimeout_lock);
4474     mutex_destroy(&ipst->ips_ip_addr_avail_lock);
4475     rw_destroy(&ipst->ips_ill_g_lock);
4477     kmem_free(ipst->ips_phyint_g_list, sizeof (phyint_list_t));
4478     ipst->ips_phyint_g_list = NULL;
4479     kmem_free(ipst->ips_ill_g_heads, sizeof (ill_g_head_t) * MAX_G_HEADS);
4480     ipst->ips_ill_g_heads = NULL;
4482     ldi_ident_release(ipst->ips_ldi_ident);
4483     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 TLLs 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_usersrc_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_mrrouter_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_req_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_rcv_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_rcv_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_rcv_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_rcv)(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_rcv_attr_t *ira)
4987 {
4988     mblk_t     *mpl;
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 = connfp->connf_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 ((m1 = dupmsg(mp)) == NULL) &&
5045         ((m1 = 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, m1, (ipha_t *)m1->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_rcv_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->ipst_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 *) (orptra + 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(orptra, orptr + shift, iph_len);
5158     if (esp_ports != 0) /* Punt up for ESP processing. */ {
5159         ipha = (ipha_t *) (orptra + 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_rcv_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_cntrlid :
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_rcvicmp)(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_rcv)(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_rcv_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_LABELLED) ||
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         *mpl;

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 (((mpl = dupmsg(mp)) == NULL) &&
5308                 ((mpl = 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, mpl, (ipha_t *)mpl->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         *mpl;

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 (((mpl = dupmsg(mp)) == NULL) &&
5423     ((mpl = 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, mpl, (ipha_t *)mpl->brptr,
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    totalen;
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     totalen = ipha->ipha_version_and_hdr_length -
5476         (uint8_t)((IP_VERSION << 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);

5478     if (totalen == 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     totalen <= 2;
5500     opt = (uchar_t *)&ipha[1];
5501     if (!is_system_labeled()) {

5503     copyall:
5504         if (!allocate) {
5505             if (totalen != 0) {
5506                 ipp->ipp_ipv4_options = opt;
5507                 ipp->ipp_ipv4_options_len = totalen;
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 (totalen == ipp->ipp_ipv4_options_len) {
5515                 bcopy(opt, ipp->ipp_ipv4_options, totalen);
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 (totalen == 0)
5525             return (0);

5527         ipp->ipp_ipv4_options = kmem_alloc(totalen, KM_NOSLEEP);
5528         if (ipp->ipp_ipv4_options == NULL)
5529             return (ENOMEM);
5530         ipp->ipp_ipv4_options_len = totalen;
5531         ipp->ipp_fields |= IPPF_IPV4_OPTIONS;
5532         bcopy(opt, ipp->ipp_ipv4_options, totalen);
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 }
5542
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 (totalen != 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 (totalen <= IPOPT_OLEN)
5558                 return (EINVAL);
5559             optlen = opt[IPOPT_OLEN];
5560             if (optlen < 2)
5561                 return (EINVAL);
5562     }
5563     if (optlen > totalen)
5564         return (EINVAL);
5565
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             totalen -= optlen;
5582             opt += optlen;
5583
5584             /* Skip padding bytes until we get to a multiple of 4 */
5585             while ((totalen & 3) != 0 && opt[0] == IPOPT_NOP) {
5586                 totalen--;
5587                 opt++;
5588             }
5589             /* Remaining as ipp_ipv4_options */
5590             goto copyall;
5591     }
5592     totalen -= optlen;
5593     opt += optlen;
5594 }
5595 /* No CIPSO found; return everything as ipp_ipv4_options */
5596 totalen = ipha->ipha_version_and_hdr_length -
5597     (uint8_t)((IP_VERSION << 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);
5598 totalen <<= 2;
5599 opt = (uchar_t *)&ipha[1];
5600 goto copyall;
5601 }
5602
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;
5614
5615     ire = ire_fhtable_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 }
5624
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;
5636
5637     ire = ire_fhtable_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 }
5646
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                 greply(q, mp);
5660                 return;
5661             }
5662             break;
5663     }
5664     freemsg(mp);
5665 }
5666
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_message_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_message_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_SRRR:
5701             case IPOPT_LSRR:
5702                 if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
5703                     ipldbg(("ip_message_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                     ipldbg(("ip_message_options: end of SR\n"));
5714                     break;
5715                 }
5716                 bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
5717                 ipldbg(("ip_message_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                     ipldbg(("ip_message_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     ipldbg(("ip_message_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_OLLEN] - 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 #ifndef 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 (0xfffffffU);

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 a conn_t as the instance data.
5981      */
5982     connp = ipcl_conn_create(IPCL_IPCONN, 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_cpuid = curproc->p_pid;
6037     /* Cache things in ixa without an extra rehold */
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_cpuid = connp->conn_cpuid;
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 appropriately. 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 checked 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             *il = (int *)invalp;
6284         int             error = 0;
6285         ip_stack_t     *ipst = connp->conn_netstack->netstack_ip;
6286         struct ip_mreq *v4_mreq;
6287         struct ipv6_mreq *v6_mreq;
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 *)il;
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 *)il;
6340     v6group = v6_mreqp->ipv6mr_multiaddr;
6341     ifaddr = INADDR_ANY;
6342     ifindex = v6_mreqp->ipv6mr_interface;
6343 } else {
6344     v4_mreqp = (struct ip_mreq *)il;
6345     IN6_INADDR_TO_V4MAPPED(&v4_mreqp->imr_multiaddr, &v6group);
6346     ifaddr = (ipaddr_t)v4_mreqp->imr_interface.s_addr;
6347     ifindex = 0;
6348 }
6349
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;
6361
6362     IN6_V4MAPPED_TO_IPADDR(&v6group, group);
6363
6364     ire = ire_fhtable_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_fhtable_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 }
6380
6381 if (!done) {
6382     error = optfn(connp, checkonly, &v6group, ifaddr, ifindex,
6383         fmode, &ipv6_all_zeros);
6384 }
6385 return (error);
6386 }
6387
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             *il = (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 *);
6413
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;
6422
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;
6430
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;
6438
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     }
6449
6450     if (mcast_opt) {
6451         gsreqp = (struct group_source_req *)il;
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;
6461
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 *)il;
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_fhtable_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_fhtable_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 = ixax->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 *mpl;
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 (mpl = ipf->ipf_mp->b_cont; mpl; mpl = mpl->b_cont) {
6670         IP_REASS_SET_START(mpl, offset);
6671         if (offset == 0) {
6672             ASSERT(ipf->ipf_nf_hdr_len != 0);
6673             offset = -ipf->ipf_nf_hdr_len;
6674         }
6675         offset += mpl->b_wptr - mpl->b_rptr;
6676         IP_REASS_SET_END(mpl, 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 interesting 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          */
6758         if (start + msgdsize(mp) !=
6759             IP_REASS_END(ipf->ipf_tail_mp)) {
6760             /*
6761              * We have two fragments both of which claim
6762              * to be the last fragment but gives conflicting
6763              * information about the whole datagram size.
6764              * Something fishy is going on. Drop the
6765              * fragment and free up the reassembly list.
6766              */
6767             return (IP_REASS_FAILED);
6768         }
6769     }
6770     /*
6771      * We shouldn't come to this code block again for this
6772      * particular fragment.
6773      */
6774     pkt_boundary_checked = B_TRUE;
6775 }
6776
6777 /* New stuff at or beyond tail? */
6778 offset = IP_REASS_END(ipf->ipf_tail_mp);
6779 if (start >= offset) {
6780     if (ipf->ipf_last_frag_seen) {
6781         /* current fragment is beyond last fragment */
6782         return (IP_REASS_FAILED);
6783     }
6784     /* Link it on end. */
6785     ipf->ipf_tail_mp->b_cont = mp;
6786     ipf->ipf_tail_mp = mp;
6787     if (more) {
6788         if (start != offset)
6789             ipf->ipf_hole_cnt++;
6790     } else if (start == offset && next_mp == NULL)
6791         ipf->ipf_hole_cnt--;
6792     continue;
6793 }

```

```

6794     }
6795     mpl = ipf->ipf_mp->b_cont;
6796     offset = IP_REASS_START(mpl);
6797     /* New stuff at the front? */
6798     if (start < offset) {
6799         if (start == 0) {
6800             if (end >= offset) {
6801                 /* Nailed the hole at the beginning. */
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 = mpl;
6812         /* Check for overlap. */
6813         while (end > offset) {
6814             if (end < IP_REASS_END(mpl)) {
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 ((mpl->b_cont &&
6823                 IP_REASS_END(mpl) !=
6824                 IP_REASS_START(mpl->b_cont) &&
6825                 end >= IP_REASS_START(mpl->b_cont)) ||
6826                 (!ipf->ipf_last_frag_seen && !more)) {
6827                 ipf->ipf_hole_cnt--;
6828             }
6829             /* Clip out mpl. */
6830             if ((mp->b_cont = mpl->b_cont) == NULL) {
6831                 /*
6832                  * After clipping out mpl, this guy
6833                  * is now hanging off the end.
6834                  */
6835                 ipf->ipf_tail_mp = mp;
6836             }
6837             IP_REASS_SET_START(mpl, 0);
6838             IP_REASS_SET_END(mpl, 0);
6839             /* Subtract byte count */
6840             ipf->ipf_count -= mpl->b_datap->db_lim -
6841                 mpl->b_datap->db_base;
6842             freeb(mpl);
6843             BUMP_MIB(ill->ill_ip_mib,
6844                 ipIfStatsReasmPartDups);
6845             mpl = mp->b_cont;
6846             if (!mpl)
6847                 break;
6848             offset = IP_REASS_START(mpl);
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 (; mpl; mpl = mpl->b_cont) {
6858         offset = IP_REASS_END(mpl);
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 (!mpl->b_cont) {
6886             /*
6887              * After trimming, this guy is now
6888              * hanging off the end.
6889              */
6890             mpl->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(mpl->b_cont))
6899         continue;
6900     /* Fill a hole */
6901     if (start > offset)
6902         ipf->ipf_hole_cnt++;
6903     mp->b_cont = mpl->b_cont;
6904     mpl->b_cont = mp;
6905     mpl = mp->b_cont;
6906     offset = IP_REASS_START(mpl);
6907     if (end >= offset) {
6908         ipf->ipf_hole_cnt--;
6909         /* Check for overlap. */
6910         while (end > offset) {
6911             if (end < IP_REASS_END(mpl)) {
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 ((mpl->b_cont &&
6925                 IP_REASS_END(mpl)

```

```

6926             != IP_REASS_START(mpl->b_cont) &&
6927             end >=
6928             IP_REASS_START(mpl->b_cont)) ||
6929             (!ipf->ipf_last_frag_seen &&
6930             !more)) {
6931                 ipf->ipf_hole_cnt--;
6932             }
6933             /* Clip out mpl. */
6934             if ((mp->b_cont = mpl->b_cont) ==
6935                 NULL) {
6936                 /*
6937                  * After clipping out mpl,
6938                  * this guy is now hanging
6939                  * off the end.
6940                  */
6941                 ipf->ipf_tail_mp = mp;
6942             }
6943             IP_REASS_SET_START(mpl, 0);
6944             IP_REASS_SET_END(mpl, 0);
6945             /* Subtract byte count */
6946             ipf->ipf_count -=
6947                 mpl->b_datap->db_lim -
6948                 mpl->b_datap->db_base;
6949             freeb(mpl);
6950             BUMP_MIB(ill->ill_ip_mib,
6951                 ipIfStatsReasmPartDups);
6952             mpl = mp->b_cont;
6953             if (!mpl)
6954                 break;
6955             offset = IP_REASS_START(mpl);
6956         }
6957     }
6958     } while (start = end, mp = next_mp);
6959 }
6960
6961 /* Fragment just processed could be the last one. Remember this fact */
6962 if (!more)
6963     ipf->ipf_last_frag_seen = B_TRUE;
6964
6965 /* Still got holes? */
6966 if (ipf->ipf_hole_cnt)
6967     return (IP_REASS_PARTIAL);
6968 /* Clean up overloaded fields to avoid upstream disasters. */
6969 for (mpl = ipf->ipf_mp->b_cont; mpl; mpl = mpl->b_cont) {
6970     IP_REASS_SET_START(mpl, 0);
6971     IP_REASS_SET_END(mpl, 0);
6972 }
6973 return (IP_REASS_COMPLETE);
6974 }
6975
6976 /*
6977  * Fragmentation reassembly. Each ILL has a hash table for
6978  * queuing packets undergoing reassembly for all IPIFs
6979  * associated with the ILL. The hash is based on the packet
6980  * IP ident field. The ILL frag hash table was allocated
6981  * as a timer block at the time the ILL was created. Whenever
6982  * there is anything on the reassembly queue, the timer will
6983  * be running. Returns the reassembled packet if reassembly completes.
6984  */
6985 mblk_t *
6986 ip_input_fragment(mblk_t *mp, ipha_t *ipha, ip_rcv_attr_t *ira)
6987 {
6988     uint32_t frag_offset_flags;
6989     mblk_t *t_mp;
6990     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      *mpl;
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 *mpl = 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             (mpl == NULL || mpl->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 postpendded data.

```

```

7058         /*
7059          * IP_ADJCKSUM_PARTIAL(mp->b_rptr + DB_CKSUMSTART(mp),
7060          * mp, mpl, 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     mpl = allocb(sizeof (*ipf), BPRI_MED);
7184     if (mpl == 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     mpl->b_cont = mp;

7196     /* Initialize the fragment header. */
7197     ipf = (ipf_t *)mpl->b_rptr;
7198     ipf->ipf_mp = mpl;
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(mpl);
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     mpl = 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 = mpl->b_cont;

7391 freeb(mpl);

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_rcv_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_rcv_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_rcv_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_rcv_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 *mpl = 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 = mpl;
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 (msgsize(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_rcv_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 += msgsize(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_rcv_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.

```

```

7652 * If a different IRE is returned, then it is held; the caller
7653 * needs to release it.
7654 * In no case is there any hold/release on the ire argument.
7655 */
7656 ire_t *
7657 ip_check_multihome(void *addr, ire_t *ire, ill_t *ill)
7658 {
7659     ire_t      *new_ire;
7660     ill_t      *ire_ill;
7661     uint_t     ifindex;
7662     ip_stack_t *ipst = ill->ill_ipst;
7663     boolean_t  strict_check = B_FALSE;

7664     /*
7665      * IPMP common case: if IRE and ILL are in the same group, there's no
7666      * issue (e.g. packet received on an underlying interface matched an
7667      * IRE_LOCAL on its associated group interface).
7668      */
7669     ASSERT(ire->ire_ill != NULL);
7670     if (IS_IN_SAME_ILLGRP(ill, ire->ire_ill))
7671         return (ire);

7672     /*
7673      * Do another ire lookup here, using the ingress ill, to see if the
7674      * interface is in a usesrc group.
7675      * As long as the ills belong to the same group, we don't consider
7676      * them to be arriving on the wrong interface. Thus, if the switch
7677      * is doing inbound load spreading, we won't drop packets when the
7678      * ip*_strict_dst_multihoming switch is on.
7679      * We also need to check for IPIF_UNNUMBERED point2point interfaces
7680      * where the local address may not be unique. In this case we were
7681      * at the mercy of the initial ire lookup and the IRE_LOCAL it
7682      * actually returned. The new lookup, which is more specific, should
7683      * only find the IRE_LOCAL associated with the ingress ill if one
7684      * exists.
7685      */
7686     if (ire->ire_ipversion == IPV4_VERSION) {
7687         if (ipst->ips_ip_strict_dst_multihoming)
7688             strict_check = B_TRUE;
7689         new_ire = ire_ftable_lookup_v4*((ipaddr_t *)addr), 0, 0,
7690             IRE_LOCAL, ill, ALL_ZONES, NULL,
7691             (MATCH_IRE_TYPE|MATCH_IRE_ILL), 0, ipst, NULL);
7692     } else {
7693         ASSERT(!IN6_IS_ADDR_MULTICAST((in6_addr_t *)addr));
7694         if (ipst->ips_ipv6_strict_dst_multihoming)
7695             strict_check = B_TRUE;
7696         new_ire = ire_ftable_lookup_v6((in6_addr_t *)addr, NULL, NULL,
7697             IRE_LOCAL, ill, ALL_ZONES, NULL,
7698             (MATCH_IRE_TYPE|MATCH_IRE_ILL), 0, ipst, NULL);
7699     }
7700     /*
7701      * If the same ire that was returned in ip_input() is found then this
7702      * is an indication that usesrc groups are in use. The packet
7703      * arrived on a different ill in the group than the one associated with
7704      * the destination address. If a different ire was found then the same
7705      * IP address must be hosted on multiple ills. This is possible with
7706      * unnumbered point2point interfaces. We switch to use this new ire in
7707      * order to have accurate interface statistics.
7708      */
7709     if (new_ire != NULL) {
7710         /* Note: held in one case but not the other? Caller handles */
7711         if (new_ire != ire)
7712             return (new_ire);
7713         /* Unchanged */
7714         ire_refrele(new_ire);
7715         return (ire);
7716     }

```

```

7718     }

7720     /*
7721      * Chase pointers once and store locally.
7722      */
7723     ASSERT(ire->ire_ill != NULL);
7724     ire_ill = ire->ire_ill;
7725     ifindex = ill->ill_usesrc_ifindex;

7727     /*
7728      * Check if it's a legal address on the 'usesrc' interface.
7729      * For IPMP data addresses the IRE_LOCAL is the upper, hence we
7730      * can just check phyint_ifindex.
7731      */
7732     if (ifindex != 0 && ifindex == ire_ill->ill_phyint->phyint_ifindex) {
7733         return (ire);
7734     }

7736     /*
7737      * If the ip*_strict_dst_multihoming switch is on then we can
7738      * only accept this packet if the interface is marked as routing.
7739      */
7740     if (!(strict_check))
7741         return (ire);

7743     if ((ill->ill_flags & ire->ire_ill->ill_flags & ILLF_ROUTER) != 0) {
7744         return (ire);
7745     }
7746     return (NULL);
7747 }

7749 /*
7750 * This function is used to construct a mac_header_info_s from a
7751 * DL_UNITDATA_IND message.
7752 * The address fields in the mhi structure points into the message,
7753 * thus the caller can't use those fields after freeing the message.
7754 *
7755 * We determine whether the packet received is a non-unicast packet
7756 * and in doing so, determine whether or not it is broadcast vs multicast.
7757 * For it to be a broadcast packet, we must have the appropriate mblk_t
7758 * hanging off the ill_t. If this is either not present or doesn't match
7759 * the destination mac address in the DL_UNITDATA_IND, the packet is deemed
7760 * to be multicast. Thus NICs that have no broadcast address (or no
7761 * capability for one, such as point to point links) cannot return as
7762 * the packet being broadcast.
7763 */
7764 void
7765 ip_dlur_to_mhi(ill_t *ill, mblk_t *mb, struct mac_header_info_s *mhip)
7766 {
7767     dl_unitdata_ind_t *ind = (dl_unitdata_ind_t *)mb->b_rptr;
7768     mblk_t *bmp;
7769     uint_t extra_offset;

7771     bzero(mhip, sizeof (struct mac_header_info_s));

7773     mhip->mhi_dsttype = MAC_ADDRTYPE_UNICAST;

7775     if (ill->ill_sap_length < 0)
7776         extra_offset = 0;
7777     else
7778         extra_offset = ill->ill_sap_length;

7780     mhip->mhi_daddr = (uchar_t *)ind + ind->dl_dest_addr_offset +
7781         extra_offset;
7782     mhip->mhi_saddr = (uchar_t *)ind + ind->dl_src_addr_offset +
7783         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 *iocpb;
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);
7925
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         ipldbg(("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 }
7973
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;
7980
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;
7999
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     }
8006
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;
8016
8017         IP_STAT(ipst, ip_db_ref);
8018
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);
8022
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     }
8041
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_INVALID;

8073     DTRACE_PROBE3(ill_dlpi, char *, "ip_rput_dlpi",
8074 char *, dl_primstr(prim), ill_t *, ill);
8075     ipldbg(("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_INVALID ||
8111             !ill_dlpi_pending(ill, reqprim)) {
8112             /* Not a DLPI message we support or expected */
8113             freemsg(mp);

```

```

8114         return;
8115     }
8116     ipldbg(("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             break;
8137         }
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 rehold 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         *mpl = 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);

8184     ipldbg(("ip_rput_dlpi_writer .."));
8185     ASSERT(ipsq->ipsq_xop == ill->ill_phyint->phyint_ipsq->ipsq_xop);
8186     ASSERT(IAM_WRITER_ILL(ill));

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;

8197     switch (dloa->dl_primitive) {
8198     case DL_ERROR_ACK:
8199         ipldbg(("ip_rput_dlpi_writer: got DL_ERROR_ACK for %s\n",
8200             dl_primstr(dlea->dl_error_primitive)));

8202     DTRACE_PROBE3(ill_dlpi, char *, "ip_rput_dlpi_writer error",
8203                 char *, dl_primstr(dlea->dl_error_primitive),
8204                 ill_t *, ill);

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_lls_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         mpl = ipsq_pending_mp_get(ipsq, &connp);
8260     if (mpl != 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             mpl = ipsq_pending_mp_get(ipsq, &connp);
8284         if (mpl != 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);
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);
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) {
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);
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_MULTICAST;
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     ipldbg(("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 (mpl is set when
8332  * ready to complete ioctl). If ill_ifname_pending_err is
8333  * set, an error occurred during plumbing (ill_ifname_pending),
8334  * so we want to report that error.
8335  *
8336  * NOTE: there are two additional 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 (mpl != 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 /*
8352  * If we're plumbing an interface and an error hasn't already
8353  * been saved, set ill_ifname_pending_err to the error passed
8354  * up. Ignore the error if log is B_FALSE (see comment above).
8355  */
8356 } else if (log && ill->ill_ifname_pending &&
8357     ill->ill_ifname_pending_err == 0) {
8358     ill->ill_ifname_pending_err = dlea->dl_unix_errno ?
8359         dlea->dl_unix_errno : ENXIO;
8360 }

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         mpl = ipsq_pending_mp_get(ipsq, &connp);
8399     if (mpl == NULL) {
8400         DTRACE_PROBE1(ip_rput_dlpi_no_mblk, ill_t *, ill);
8401         break;
8402     }
8403     /*
8404      * mpl 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     ipldbg(("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, mpl, 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             mpl = 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     }
8464
8465     if ((err == 0) && (ill->ill_up_ipifs)) {
8466         err = ill_up_ipifs(ill, q, mpl);
8467         if (err == EINPROGRESS) {
8468             freemsg(mp);
8469             return;
8470         }
8471     }
8472
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, mpl);
8486             if (err == EINPROGRESS) {
8487                 freemsg(mp);
8488                 return;
8489             }
8490         }
8491     }
8492     break;
8493
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;
8497
8498         switch (notify->dl_notification) {
8499             case DL_NOTE_PHYS_ADDR:
8500                 err = ill_set_phys_addr(ill, mp);
8501                 break;
8502
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;
8511
8512             case DL_NOTE_FASTPATH_FLUSH:
8513                 nce_flush(ill, B_FALSE);
8514                 break;
8515
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;
8543
8544                 if (!(ill->ill_flags & ILLF_FIXEDMTU)) {
8545                     ill->ill_mtu = ill->ill_current_frag;
8546
8547                     /*
8548                      * If ill_user_mtu was set (via
8549                      * SIOCSLIPLNKINFO), 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;
8554
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;
8558
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                 }
8573
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 assoc's 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_rprtr;
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 SIOCCLIFNAME,
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 paddrln, paddroff;
8709     uint8_t *addr;

8711     paddrreq = ill->ill_phys_addr_pend;
8712     paddrln = ((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], paddrln);
8725         ill->ill_token_length = paddrln;
8726         break;
8727     } else if (paddrreq == DL_IPV6_LINK_LAYER_ADDR) {
8728         ASSERT(ill->ill_nd_llsmp == NULL);
8729         ill_set_ndmp(ill, mp, paddroff, paddrln);
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         mpl = ipsq_pending_mp_get(ipsq, &connp);
8751     if (mpl != 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 = (paddrln == 0 ? NULL : addr);
8768     mp = NULL;

8770     /*
8771     * If paddrln or ill_phys_addr_length is zero, the DLPI
8772     * provider doesn't support physical addresses. We check both
8773     * paddrln and ill_phys_addr_length because sPPP (PPP) does

```

```

8774     * not have physical addresses, but historically advertises a
8775     * physical address length of 0 in its DL_INFO_ACK, but 6 in
8776     * its DL_PHYS_ADDR_ACK.
8777     */
8778     if (paddrln == 0 || ill->ill_phys_addr_length == 0) {
8779         ill->ill_phys_addr = NULL;
8780     } else if (paddrln != ill->ill_phys_addr_length) {
8781         ip0dbg("DL_PHYS_ADDR_ACK: got addrln %d, expected %d",
8782             paddrln, ill->ill_phys_addr_length);
8783         err = EINVAL;
8784         break;
8785     }

8787     if (ill->ill_nd_llsmp == 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, paddrln);
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 (mpl == NULL)
8828         return;

8830     /*
8831     * The operation must complete without EINPROGRESS since
8832     * ipsq_pending_mp_get() has removed the mblk (mpl). 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     ipldbg(("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             ipldbg(("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;

```



```

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         gethrtime(&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_rcv_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         ipldbg(("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     ipldbg(("ip_input_local_options: not end of SR\n"));
9251     if (optval == IPOPT_SRRR) {
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         ipldbg(("
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 room */
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         break;
9303     default:
9304         /*
9305          * ip_put_options should have already
9306          * dropped this packet.
9307          */
9308         cmn_err(CE_PANIC, "ip_input_local_options: "
9309             "unknown IT - bug in ip_input_options?\n");
9310         return (B_TRUE); /* Keep "lint" happy */
9311     }
9312     if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
9313         /* Increase overflow counter */
9314         off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
9315         opt[IPOPT_POS_OV_FLG] =
9316             (uint8_t)((opt[IPOPT_POS_OV_FLG] & 0x0F) |
9317                 (off << 4));
9318         break;
9319     }
9320     off = opt[IPOPT_OFFSET] - 1;
9321     switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9322     case IPOPT_TS_PRESPEC:
9323     case IPOPT_TS_PRESPEC_RFC791:
9324     case IPOPT_TS_TSANDADDR:
9325         /* Pick a reasonable addr on the outbound if */
9326         if (ip_select_source_v4(ill, INADDR_ANY,
9327             ipha->ipha_dst, INADDR_ANY, ALL_ZONES, ipst,
9328             &ifaddr, NULL, NULL) != 0) {
9329             /* No source! Shouldn't happen */
9330             ifaddr = INADDR_ANY;
9331         }
9332         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9333         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9334         /* FALLTHRU */
9335     case IPOPT_TS_TSONLY:
9336         off = opt[IPOPT_OFFSET] - 1;
9337         /* Compute # of milliseconds since midnight */
9338         gethrestime(&now);
9339         ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
9340             now.tv_nsec / (NANOSEC / MILLISEC);
9341         bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
9342         opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
9343         break;
9344     }
9345     }
9346     }
9347     }
9348     return (B_TRUE);
9349 }
9350 bad_src_route:
9351     /* make sure we clear any indication of a hardware checksum */
9352     DB_CKSUMFLAGS(mp) = 0;
9353     ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
9354     icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
9355     return (B_FALSE);
9356 }
9357
9358 /*
9359  * Process IP options in an inbound packet. Always returns the nexthop.
9360  * Normally this is the passed in nexthop, but if there is an option
9361  * that effects the nexthop (such as a source route) that will be returned.
9362  * Sets *errorp if there is an error, in which case an ICMP error has been sent
9363  * and mp freed.
9364  */
9365 ipaddr_t
9366 ip_input_options(ipha_t *ipha, ipaddr_t dst, mblk_t *mp,

```

```

9368     ip_rcv_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                 }
9410             if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9411                 ip1dbg(("ip_input_options: bad option offset\n"));
9412                 code = (char *)&opt[IPOPT_OLEN] -
9413                       (char *)ipha;
9414                 goto param_prob;
9415             }
9416             off = opt[IPOPT_OFFSET];
9417             off--;
9418             redo_srr:
9419             if (optlen < IP_ADDR_LEN ||
9420                 off > optlen - IP_ADDR_LEN) {
9421                 /* End of source route */
9422                 ip1dbg(("ip_input_options: end of SR\n"));
9423                 break;
9424             }
9425             bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9426             ip1dbg(("ip_input_options: next hop 0x%x\n",
9427                    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_fhtable_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_refere(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(("ip_input_options: bad option offset\n"));
9473                     code = (char *)&opt[IPOPT_OLEN] -
9474                           (char *)ipha;
9475                     goto param_prob;
9476                 }
9477                 break;
9478             case IPOPT_TS:
9479                 /*
9480                  * Verify that length >= 5 and that there is either
9481                  * room for another timestamp or that the overflow
9482                  * counter is not maxed out.
9483                  */
9484                 code = (char *)&opt[IPOPT_OLEN] - (char *)ipha;
9485                 if (optlen < IPOPT_MINLEN_IT) {
9486                     goto param_prob;
9487                 }
9488                 if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9489                     ip1dbg(("ip_input_options: bad option offset\n"));
9490                     code = (char *)&opt[IPOPT_OFFSET] -
9491                           (char *)ipha;
9492                     goto param_prob;
9493                 }
9494                 switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9495                     case IPOPT_TS_TSONLY:
9496                         off = IPOPT_TS_TIMELEN;
9497                         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 }
9521
9522 if ((opts.ipoptp_flags & IPOPTP_ERROR) == 0) {
9523     return (dst);
9524 }
9525
9526 ipldbg(("ip_input_options: error processing IP options."));
9527 code = (char *)&opt[IPOPT_OFFSET] - (char *)ipha;
9528
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);
9536
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 }
9545
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 mfctl)
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;
9583
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;
9591
9592     if (mpctl == NULL || mpctl->b_cont == NULL) {
9593         return (0);
9594     }
9595
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     }
9610
9611     if (level != MIB2_TCP) {
9612         if ((mpctl = udp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9613             return (1);
9614         }
9615     }
9616
9617     if (level != MIB2_UDP) {
9618         if ((mpctl = tcp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9619             return (1);
9620         }
9621     }
9622
9623     if ((mpctl = ip_snmp_get_mib2_ip_traffic_stats(q, mpctl,
9624         ipst, legacy_req)) == NULL) {
9625         return (1);
9626     }
9627
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     }
9694     if ((mpctl = ip_snmp_get_mib2_ip_dce(q, mpctl, ipst)) == NULL) {
9695         return (1);
9696     }
9697     freemsg(mpctl);

```

```

9698         return (1);
9699     }

9701     /* Get global (legacy) IPv4 statistics */
9702     static mblk_t *
9703     ip_snmp_get_mib2_ip(queue_t *q, mblk_t *mpctl, mib2_ipIfStatsEntry_t *ipmib,
9704         ip_stack_t *ipst, boolean_t legacy_req)
9705     {
9706         mib2_ip_t          old_ip_mib;
9707         struct opthdr      *optp;
9708         mblk_t             *mp2ctl;
9709         mib2_ipAddrEntry_t mae;

9711         /*
9712          * make a copy of the original message
9713          */
9714         mp2ctl = copymsg(mpctl);

9716         /* fixed length IP structure... */
9717         optp = (struct opthdr *)&mpctl->b_rptr[ sizeof (struct T_optmgmt_ack) ];
9718         optp->level = MIB2_IP;
9719         optp->name = 0;
9720         SET_MIB(old_ip_mib.ipForwarding,
9721             (WE_ARE_FORWARDING(ipst) ? 1 : 2));
9722         SET_MIB(old_ip_mib.ipDefaultTTL,
9723             (uint32_t)ipst->ips_ip_def_ttl);
9724         SET_MIB(old_ip_mib.ipReasmTimeout,
9725             ipst->ips_ip_reassembly_timeout);
9726         SET_MIB(old_ip_mib.ipAddrEntrySize,
9727             (legacy_req) ? LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t) :
9728             sizeof (mib2_ipAddrEntry_t));
9729         SET_MIB(old_ip_mib.ipRouteEntrySize,
9730             sizeof (mib2_ipRouteEntry_t));
9731         SET_MIB(old_ip_mib.ipNetToMediaEntrySize,
9732             sizeof (mib2_ipNetToMediaEntry_t));
9733         SET_MIB(old_ip_mib.ipMemberEntrySize, sizeof (ip_member_t));
9734         SET_MIB(old_ip_mib.ipGroupSourceEntrySize, sizeof (ip_grpsrc_t));
9735         SET_MIB(old_ip_mib.ipRouteAttributeSize,
9736             sizeof (mib2_ipAttributeEntry_t));
9737         SET_MIB(old_ip_mib.transportMLPSize, sizeof (mib2_transportMLPEntry_t));
9738         SET_MIB(old_ip_mib.ipDestEntrySize, sizeof (dest_cache_entry_t));

9740         /*
9741          * Grab the statistics from the new IP MIB
9742          */
9743         SET_MIB(old_ip_mib.ipInReceives,
9744             (uint32_t)ipmib->ipIfStatsHCInReceives);
9745         SET_MIB(old_ip_mib.ipInHdrErrors, ipmib->ipIfStatsInHdrErrors);
9746         SET_MIB(old_ip_mib.ipInAddrErrors, ipmib->ipIfStatsInAddrErrors);
9747         SET_MIB(old_ip_mib.ipForwDatagrams,
9748             (uint32_t)ipmib->ipIfStatsHCOutForwDatagrams);
9749         SET_MIB(old_ip_mib.ipInUnknownProtos,
9750             ipmib->ipIfStatsInUnknownProtos);
9751         SET_MIB(old_ip_mib.ipInDiscards, ipmib->ipIfStatsInDiscards);
9752         SET_MIB(old_ip_mib.ipInDelivers,
9753             (uint32_t)ipmib->ipIfStatsHCInDelivers);
9754         SET_MIB(old_ip_mib.ipOutRequests,
9755             (uint32_t)ipmib->ipIfStatsHCOutRequests);
9756         SET_MIB(old_ip_mib.ipOutDiscards, ipmib->ipIfStatsOutDiscards);
9757         SET_MIB(old_ip_mib.ipOutNoRoutes, ipmib->ipIfStatsOutNoRoutes);
9758         SET_MIB(old_ip_mib.ipReasmReqds, ipmib->ipIfStatsReasmReqds);
9759         SET_MIB(old_ip_mib.ipReasmOKs, ipmib->ipIfStatsReasmOKs);
9760         SET_MIB(old_ip_mib.ipReasmFails, ipmib->ipIfStatsReasmFails);
9761         SET_MIB(old_ip_mib.ipFragOKs, ipmib->ipIfStatsOutFragOKs);
9762         SET_MIB(old_ip_mib.ipFragFails, ipmib->ipIfStatsOutFragFails);
9763         SET_MIB(old_ip_mib.ipFragCreates, ipmib->ipIfStatsOutFragCreates);

```

```

9765 /* ipRoutingDiscards is not being used */
9766 SET_MIB(old_ip_mib.ipRoutingDiscards, 0);
9767 SET_MIB(old_ip_mib.tcpInErrs, ipmib->tcpIfStatsInErrs);
9768 SET_MIB(old_ip_mib.udpNoPorts, ipmib->udpIfStatsNoPorts);
9769 SET_MIB(old_ip_mib.ipInCksumErrs, ipmib->ipIfStatsInCksumErrs);
9770 SET_MIB(old_ip_mib.ipReasmDuplicates,
9771 ipmib->ipIfStatsReasmDuplicates);
9772 SET_MIB(old_ip_mib.ipReasmPartDups, ipmib->ipIfStatsReasmPartDups);
9773 SET_MIB(old_ip_mib.ipForwProhibits, ipmib->ipIfStatsForwProhibits);
9774 SET_MIB(old_ip_mib.udpInCksumErrs, ipmib->udpIfStatsInCksumErrs);
9775 SET_MIB(old_ip_mib.udpInOverflows, ipmib->udpIfStatsInOverflows);
9776 SET_MIB(old_ip_mib.rawipInOverflows,
9777 ipmib->rawipIfStatsInOverflows);

9779 SET_MIB(old_ip_mib.ipsecInSucceeded, ipmib->ipsecIfStatsInSucceeded);
9780 SET_MIB(old_ip_mib.ipsecInFailed, ipmib->ipsecIfStatsInFailed);
9781 SET_MIB(old_ip_mib.ipInIPv6, ipmib->ipIfStatsOutWrongIPVersion);
9782 SET_MIB(old_ip_mib.ipOutIPv6, ipmib->ipIfStatsOutWrongIPVersion);
9783 SET_MIB(old_ip_mib.ipOutSwitchIPv6,
9784 ipmib->ipIfStatsOutSwitchIPv6);

9786 if (!snmp_append_data(mpctl->b_cont, (char *)&old_ip_mib,
9787 (int)sizeof (old_ip_mib))) {
9788     ipldbg(("ip_snmp_get_mib2_ip: failed to allocate %u bytes\n",
9789 (uint_t)sizeof (old_ip_mib)));
9790 }

9792 optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
9793 ip3dbg(("ip_snmp_get_mib2_ip: level %d, name %d, len %d\n",
9794 (int)optp->level, (int)optp->name, (int)optp->len));
9795 greply(q, mpctl);
9796 return (mp2ctl);
9797 }

9799 /* Per interface IPv4 statistics */
9800 static mblk_t *
9801 ip_snmp_get_mib2_ip_traffic_stats(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
9802 boolean_t legacy_req)
9803 {
9804     struct ophdr *optp;
9805     mblk_t *mp2ctl;
9806     ill_t *ill;
9807     ill_walk_context_t ctx;
9808     mblk_t *mp_tail = NULL;
9809     mib2_ipIfStatsEntry_t global_ip_mib;
9810     mib2_ipAddrEntry_t mae;

9812 /*
9813  * Make a copy of the original message
9814  */
9815     mp2ctl = copymsg(mpctl);

9817     optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optgmt_ack)];
9818     optp->level = MIB2_IP;
9819     optp->name = MIB2_IP_TRAFFIC_STATS;
9820     /* Include "unknown interface" ip_mib */
9821     ipst->ips_ip_mib.ipIfStatsIPVersion = MIB2_INETADDRESSSTYPE_ipv4;
9822     ipst->ips_ip_mib.ipIfStatsIfIndex =
9823     MIB2_UNKNOWN_INTERFACE; /* Flag to netstat */
9824     SET_MIB(ipst->ips_ip_mib.ipIfStatsForwarding,
9825 (ipst->ips_ip_forwarding ? 1 : 2));
9826     SET_MIB(ipst->ips_ip_mib.ipIfStatsDefaultTTL,
9827 (uint32_t)ipst->ips_ip_def_ttl);
9828     SET_MIB(ipst->ips_ip_mib.ipIfStatsEntrySize,
9829 sizeof (mib2_ipIfStatsEntry_t));

```

```

9830     SET_MIB(ipst->ips_ip_mib.ipIfStatsAddrEntrySize,
9831     sizeof (mib2_ipAddrEntry_t));
9832     SET_MIB(ipst->ips_ip_mib.ipIfStatsRouteEntrySize,
9833     sizeof (mib2_ipRouteEntry_t));
9834     SET_MIB(ipst->ips_ip_mib.ipIfStatsNetToMediaEntrySize,
9835     sizeof (mib2_ipNetToMediaEntry_t));
9836     SET_MIB(ipst->ips_ip_mib.ipIfStatsMemberEntrySize,
9837     sizeof (ip_member_t));
9838     SET_MIB(ipst->ips_ip_mib.ipIfStatsGroupSourceEntrySize,
9839     sizeof (ip_grpsrc_t));

9841     bcopy(&ipst->ips_ip_mib, &global_ip_mib, sizeof (global_ip_mib));

9843     if (legacy_req) {
9844         SET_MIB(global_ip_mib.ipIfStatsAddrEntrySize,
9845         LEGACY_MIB_SIZE(mae, mib2_ipAddrEntry_t));
9846     }

9848     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
9849 (char *)&global_ip_mib, (int)sizeof (global_ip_mib))) {
9850         ipldbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9851 "failed to allocate %u bytes\n",
9852 (uint_t)sizeof (global_ip_mib)));
9853     }

9855     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
9856     ill = ILL_START_WALK_V4(&ctx, ipst);
9857     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
9858         ill->ill_ip_mib->ipIfStatsIfIndex =
9859         ill->ill_phyint->phyint_ifindex;
9860         SET_MIB(ill->ill_ip_mib->ipIfStatsForwarding,
9861 (ipst->ips_ip_forwarding ? 1 : 2));
9862         SET_MIB(ill->ill_ip_mib->ipIfStatsDefaultTTL,
9863 (uint32_t)ipst->ips_ip_def_ttl);

9865         ip_mib2_add_ip_stats(&global_ip_mib, ill->ill_ip_mib);
9866         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
9867 (char *)ill->ill_ip_mib,
9868 (int)sizeof (*ill->ill_ip_mib))) {
9869             ipldbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9870 "failed to allocate %u bytes\n",
9871 (uint_t)sizeof (*ill->ill_ip_mib)));
9872         }
9873     }
9874     rw_exit(&ipst->ips_ill_g_lock);

9876     optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
9877     ip3dbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9878 "level %d, name %d, len %d\n",
9879 (int)optp->level, (int)optp->name, (int)optp->len));
9880     greply(q, mpctl);

9882     if (mp2ctl == NULL)
9883         return (NULL);

9885     return (ip_snmp_get_mib2_ip(q, mp2ctl, &global_ip_mib, ipst,
9886     legacy_req));
9887 }

9889 /* Global IPv4 ICMP statistics */
9890 static mblk_t *
9891 ip_snmp_get_mib2_icmp(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9892 {
9893     struct ophdr *optp;
9894     mblk_t *mp2ctl;

```

```

9896 /*
9897  * Make a copy of the original message
9898  */
9899 mp2ctl = copymsg(mpctl);

9901 optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9902 optp->level = MIB2_ICMP;
9903 optp->name = 0;
9904 if (!snmp_append_data(mpctl->b_cont, (char *)&ipst->ips_icmp_mib,
9905 (int)sizeof (ipst->ips_icmp_mib)) {
9906     ipldbg(("ip_snmp_get_mib2_icmp: failed to allocate %u bytes\n",
9907 (uint_t)sizeof (ipst->ips_icmp_mib)));
9908 }
9909 optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
9910 ip3dbg(("ip_snmp_get_mib2_icmp: level %d, name %d, len %d\n",
9911 (int)optp->level, (int)optp->name, (int)optp->len));
9912 greply(q, mpctl);
9913 return (mp2ctl);
9914 }

9916 /* Global IPv4 IGMP statistics */
9917 static mblk_t *
9918 ip_snmp_get_mib2_igmp(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9919 {
9920     struct ophdr      *optp;
9921     mblk_t            *mp2ctl;

9923 /*
9924  * make a copy of the original message
9925  */
9926 mp2ctl = copymsg(mpctl);

9928 optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9929 optp->level = EXPER_IGMP;
9930 optp->name = 0;
9931 if (!snmp_append_data(mpctl->b_cont, (char *)&ipst->ips_igmpstat,
9932 (int)sizeof (ipst->ips_igmpstat)) {
9933     ipldbg(("ip_snmp_get_mib2_igmp: failed to allocate %u bytes\n",
9934 (uint_t)sizeof (ipst->ips_igmpstat)));
9935 }
9936 optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
9937 ip3dbg(("ip_snmp_get_mib2_igmp: level %d, name %d, len %d\n",
9938 (int)optp->level, (int)optp->name, (int)optp->len));
9939 greply(q, mpctl);
9940 return (mp2ctl);
9941 }

9943 /* Global IPv4 Multicast Routing statistics */
9944 static mblk_t *
9945 ip_snmp_get_mib2_multi(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9946 {
9947     struct ophdr      *optp;
9948     mblk_t            *mp2ctl;

9950 /*
9951  * make a copy of the original message
9952  */
9953 mp2ctl = copymsg(mpctl);

9955 optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9956 optp->level = EXPER_DVMRP;
9957 optp->name = 0;
9958 if (!ip_mroute_stats(mpctl->b_cont, ipst)) {
9959     ip0dbg(("ip_mroute_stats: failed\n"));
9960 }
9961 optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);

```

```

9962     ip3dbg(("ip_snmp_get_mib2_multi: level %d, name %d, len %d\n",
9963 (int)optp->level, (int)optp->name, (int)optp->len));
9964 greply(q, mpctl);
9965 return (mp2ctl);
9966 }

9968 /* IPv4 address information */
9969 static mblk_t *
9970 ip_snmp_get_mib2_ip_addr(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
9971 boolean_t legacy_req)
9972 {
9973     struct ophdr      *optp;
9974     mblk_t            *mp2ctl;
9975     mblk_t            *mp_tail;
9976     ill_t             *ill;
9977     ipif_t            *ipif;
9978     uint_t            bitval;
9979     mib2_ipAddrEntry_t mae;
9980     size_t            mae_size;
9981     zoneid_t          zoneid;
9982     ill_walk_context_t ctx;

9984 /*
9985  * make a copy of the original message
9986  */
9987 mp2ctl = copymsg(mpctl);

9989 mae_size = (legacy_req) ? LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t) :
9990 sizeof (mib2_ipAddrEntry_t);

9992 /* ipAddrEntryTable */

9994 optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9995 optp->level = MIB2_IP;
9996 optp->name = MIB2_IP_ADDR;
9997 zoneid = Q_TO_CONN(q)->conn_zoneid;

9999 rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10000     ill = ILL_START_WALK_V4(&ctx, ipst);
10001     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10002         for (ipif = ill->ill_ipif; ipif != NULL;
10003             ipif = ipif->ipif_next) {
10004             if (ipif->ipif_zoneid != zoneid &&
10005                 ipif->ipif_zoneid != ALL_ZONES)
10006                 continue;
10007             /* Sum of count from dead IRE_LO* and our current */
10008             mae.ipAdEntInfo.ae_ibcnt = ipif->ipif_ib_pkt_count;
10009             if (ipif->ipif_ire_local != NULL) {
10010                 mae.ipAdEntInfo.ae_ibcnt +=
10011                     ipif->ipif_ire_local->ire_ib_pkt_count;
10012             }
10013             mae.ipAdEntInfo.ae_obcnt = 0;
10014             mae.ipAdEntInfo.ae_focnt = 0;

10016             ipif_get_name(ipif, mae.ipAdEntIfIndex.o_bytes,
10017 OCTET_LENGTH);
10018             mae.ipAdEntIfIndex.o_length =
10019                 mi_strlen(mae.ipAdEntIfIndex.o_bytes);
10020             mae.ipAdEntAddr = ipif->ipif_lcl_addr;
10021             mae.ipAdEntNetMask = ipif->ipif_net_mask;
10022             mae.ipAdEntInfo.ae_subnet = ipif->ipif_subnet;
10023             mae.ipAdEntInfo.ae_subnet_len =
10024                 ip_mask_to_plen(ipif->ipif_net_mask);
10025             mae.ipAdEntInfo.ae_src_addr = ipif->ipif_lcl_addr;
10026             for (bitval = 1;
10027                 bitval &&

```

```

10028         !(bitval & ipif->ipif_brd_addr);
10029         bitval <<= 1)
10030         noop;
10031         mae.ipAdEntBcastAddr = bitval;
10032         mae.ipAdEntReasmMaxSize = IP_MAXPACKET;
10033         mae.ipAdEntInfo.ae_mtu = ipif->ipif_ill->ill_mtu;
10034         mae.ipAdEntInfo.ae_metric = ipif->ipif_ill->ill_metric;
10035         mae.ipAdEntInfo.ae_broadcast_addr =
10036         ipif->ipif_brd_addr;
10037         mae.ipAdEntInfo.ae_pp_dst_addr =
10038         ipif->ipif_pp_dst_addr;
10039         mae.ipAdEntInfo.ae_flags = ipif->ipif_flags |
10040         ill->ill_flags | ill->ill_phyint->phyint_flags;
10041         mae.ipAdEntRetransmitTime =
10042         ill->ill_reachable_retrans_time;

10044         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10045         (char *)&mae, (int)mae_size)) {
10046             ipldbg(("ip_snmp_get_mib2_ip_addr: failed to "
10047             "allocate %u bytes\n", (uint_t)mae_size));
10048         }
10049     }
10050 }
10051 rw_exit(&ipst->ips_ill_g_lock);

10053 otp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10054 ip3dbg(("ip_snmp_get_mib2_ip_addr: level %d, name %d, len %d\n",
10055 (int)otp->level, (int)otp->name, (int)otp->len));
10056 qreply(q, mpctl);
10057 return (mp2ctl);
10058 }

10060 /* IPv6 address information */
10061 static mblk_t *
10062 ip_snmp_get_mib2_ip6_addr(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
10063 boolean_t legacy_req)
10064 {
10065     struct ophdr          *optp;
10066     mblk_t                *mp2ctl;
10067     mblk_t                *mp_tail = NULL;
10068     ill_t                 *ill;
10069     ipif_t                *ipif;
10070     mib2_ipv6AddrEntry_t  mae6;
10071     size_t                mae6_size;
10072     zoneid_t              zoneid;
10073     ill_walk_context_t    ctx;

10075     /*
10076      * make a copy of the original message
10077      */
10078     mp2ctl = copymsg(mpctl);

10080     mae6_size = (legacy_req) ?
10081     LEGACY_MIB_SIZE(&mae6, mib2_ipv6AddrEntry_t) :
10082     sizeof(mib2_ipv6AddrEntry_t);

10084     /* ipv6AddrEntryTable */

10086     otp = (struct ophdr *)&mpctl->b_rptr[sizeof(struct T_optgmt_ack)];
10087     otp->level = MIB2_IP6;
10088     otp->name = MIB2_IP6_ADDR;
10089     zoneid = Q_TO_CONN(q)->conn_zoneid;

10091     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10092     ill = ILL_START_WALK_V6(&ctx, ipst);
10093     for (; ill != NULL; ill = ill_next(&ctx, ill)) {

```

```

10094         for (ipif = ill->ill_ipif; ipif != NULL;
10095         ipif = ipif->ipif_next) {
10096             if (ipif->ipif_zoneid != zoneid &&
10097             ipif->ipif_zoneid != ALL_ZONES)
10098                 continue;
10099             /* Sum of count from dead IRE_LO* and our current */
10100             mae6.ipv6AddrInfo.ae_ibcnt = ipif->ipif_ib_pkt_count;
10101             if (ipif->ipif_ire_local != NULL) {
10102                 mae6.ipv6AddrInfo.ae_ibcnt +=
10103                 ipif->ipif_ire_local->ire_ib_pkt_count;
10104             }
10105             mae6.ipv6AddrInfo.ae_obcnt = 0;
10106             mae6.ipv6AddrInfo.ae_focnt = 0;

10108             ipif_get_name(ipif, mae6.ipv6AddrIfIndex.o_bytes,
10109             OCTET_LENGTH);
10110             mae6.ipv6AddrIfIndex.o_length =
10111             mi_strlen(mae6.ipv6AddrIfIndex.o_bytes);
10112             mae6.ipv6AddrAddress = ipif->ipif_v6lcl_addr;
10113             mae6.ipv6AddrPfxLength =
10114             ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
10115             mae6.ipv6AddrInfo.ae_subnet = ipif->ipif_v6subnet;
10116             mae6.ipv6AddrInfo.ae_subnet_len =
10117             mae6.ipv6AddrPfxLength;
10118             mae6.ipv6AddrInfo.ae_src_addr = ipif->ipif_v6lcl_addr;

10120             /* Type: stateless(1), stateful(2), unknown(3) */
10121             if (ipif->ipif_flags & IPIF_ADDRCONF)
10122                 mae6.ipv6AddrType = 1;
10123             else
10124                 mae6.ipv6AddrType = 2;
10125             /* Anycast: true(1), false(2) */
10126             if (ipif->ipif_flags & IPIF_ANYCAST)
10127                 mae6.ipv6AddrAnycastFlag = 1;
10128             else
10129                 mae6.ipv6AddrAnycastFlag = 2;

10131             /*
10132              * Address status: preferred(1), deprecated(2),
10133              * invalid(3), inaccessible(4), unknown(5)
10134              */
10135             if (ipif->ipif_flags & IPIF_NOLOCAL)
10136                 mae6.ipv6AddrStatus = 3;
10137             else if (ipif->ipif_flags & IPIF_DEPRECATED)
10138                 mae6.ipv6AddrStatus = 2;
10139             else
10140                 mae6.ipv6AddrStatus = 1;
10141             mae6.ipv6AddrInfo.ae_mtu = ipif->ipif_ill->ill_mtu;
10142             mae6.ipv6AddrInfo.ae_metric =
10143             ipif->ipif_ill->ill_metric;
10144             mae6.ipv6AddrInfo.ae_pp_dst_addr =
10145             ipif->ipif_v6pp_dst_addr;
10146             mae6.ipv6AddrInfo.ae_flags = ipif->ipif_flags |
10147             ill->ill_flags | ill->ill_phyint->phyint_flags;
10148             mae6.ipv6AddrReasmMaxSize = IP_MAXPACKET;
10149             mae6.ipv6AddrIdentifier = ill->ill_token;
10150             mae6.ipv6AddrIdentifierLen = ill->ill_token_length;
10151             mae6.ipv6AddrReachableTime = ill->ill_reachable_time;
10152             mae6.ipv6AddrRetransmitTime =
10153             ill->ill_reachable_retrans_time;
10154             if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10155             (char *)&mae6, (int)mae6_size)) {
10156                 ipldbg(("ip_snmp_get_mib2_ip6_addr: failed to "
10157                 "allocate %u bytes\n",
10158                 (uint_t)mae6_size));
10159             }

```

```

10160     }
10161     }
10162     rw_exit(&ipst->ips_ill_g_lock);

10164     optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
10165     ip3dbg(("ip_snmp_get_mib2_ip6_addr: level %d, name %d, len %d\n",
10166           (int)optp->level, (int)optp->name, (int)optp->len));
10167     qreply(q, mpctl);
10168     return (mp2ctl);
10169 }

10171 /* IPv4 multicast group membership. */
10172 static mblk_t *
10173 ip_snmp_get_mib2_ip_group_mem(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10174 {
10175     struct ophdr      *optp;
10176     mblk_t            *mp2ctl;
10177     ill_t             *ill;
10178     ipif_t            *ipif;
10179     ilm_t             *ilm;
10180     ip_member_t       ipm;
10181     mblk_t            *mp_tail = NULL;
10182     ill_walk_context_t ctx;
10183     zoneid_t          zoneid;

10185     /*
10186      * make a copy of the original message
10187      */
10188     mp2ctl = copymsg(mpctl);
10189     zoneid = Q_TO_CONN(q)->conn_zoneid;

10191     /* ipGroupMember table */
10192     optp = (struct ophdr *)&mpctl->b_rptr[
10193           sizeof(struct T_optmgmt_ack)];
10194     optp->level = MIB2_IP;
10195     optp->name = EXPER_IP_GROUP_MEMBERSHIP;

10197     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10198     ill = ILL_START_WALK_V4(&ctx, ipst);
10199     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10200         /* Make sure the ill isn't going away. */
10201         if (!ill_check_and_refhold(ill))
10202             continue;
10203         rw_exit(&ipst->ips_ill_g_lock);
10204         rw_enter(&ill->ill_mcast_lock, RW_READER);
10205         for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10206             if (ilm->ilm_zoneid != zoneid &&
10207                 ilm->ilm_zoneid != ALL_ZONES)
10208                 continue;

10210             /* Is there an ipif for ilm_ifaddr? */
10211             for (ipif = ill->ill_ipif; ipif != NULL;
10212                 ipif = ipif->ipif_next) {
10213                 if (!IPIF_IS_CONDEMNED(ipif) &&
10214                     ipif->ipif_lcl_addr == ilm->ilm_ifaddr &&
10215                     ilm->ilm_ifaddr != INADDR_ANY)
10216                     break;
10217             }
10218             if (ipif != NULL) {
10219                 ipif_get_name(ipif,
10220                             ipm.ipGroupMemberIfIndex.o_bytes,
10221                             OCTET_LENGTH);
10222             } else {
10223                 ill_get_name(ill,
10224                             ipm.ipGroupMemberIfIndex.o_bytes,
10225                             OCTET_LENGTH);

```

```

10226     }
10227     ipm.ipGroupMemberIfIndex.o_length =
10228     mi_strlen(ipm.ipGroupMemberIfIndex.o_bytes);

10230     ipm.ipGroupMemberAddress = ilm->ilm_addr;
10231     ipm.ipGroupMemberRefCnt = ilm->ilm_refcnt;
10232     ipm.ipGroupMemberFilterMode = ilm->ilm_fmode;
10233     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10234                           (char *)&ipm, (int)sizeof(ipm))) {
10235         ip1dbg(("ip_snmp_get_mib2_ip_group: "
10236               "failed to allocate %u bytes\n",
10237               (uint_t)sizeof(ipm)));
10238     }
10239     }
10240     rw_exit(&ill->ill_mcast_lock);
10241     ill_refrele(ill);
10242     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10243 }
10244 rw_exit(&ipst->ips_ill_g_lock);
10245 optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
10246 ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10247       (int)optp->level, (int)optp->name, (int)optp->len));
10248 qreply(q, mpctl);
10249 return (mp2ctl);
10250 }

10252 /* IPv6 multicast group membership. */
10253 static mblk_t *
10254 ip_snmp_get_mib2_ip6_group_mem(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10255 {
10256     struct ophdr      *optp;
10257     mblk_t            *mp2ctl;
10258     ill_t             *ill;
10259     ilm_t             *ilm;
10260     ipv6_member_t     ipm6;
10261     mblk_t            *mp_tail = NULL;
10262     ill_walk_context_t ctx;
10263     zoneid_t          zoneid;

10265     /*
10266      * make a copy of the original message
10267      */
10268     mp2ctl = copymsg(mpctl);
10269     zoneid = Q_TO_CONN(q)->conn_zoneid;

10271     /* ip6GroupMember table */
10272     optp = (struct ophdr *)&mpctl->b_rptr[sizeof(struct T_optmgmt_ack)];
10273     optp->level = MIB2_IP6;
10274     optp->name = EXPER_IP6_GROUP_MEMBERSHIP;

10276     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10277     ill = ILL_START_WALK_V6(&ctx, ipst);
10278     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10279         /* Make sure the ill isn't going away. */
10280         if (!ill_check_and_refhold(ill))
10281             continue;
10282         rw_exit(&ipst->ips_ill_g_lock);
10283         /*
10284          * Normally we don't have any members on under IPMP interfaces.
10285          * We report them as a debugging aid.
10286          */
10287         rw_enter(&ill->ill_mcast_lock, RW_READER);
10288         ipm6.ipv6GroupMemberIfIndex = ill->ill_phyint->phyint_ifindex;
10289         for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10290             if (ilm->ilm_zoneid != zoneid &&
10291                 ilm->ilm_zoneid != ALL_ZONES)

```

```

10292         continue;          /* not this zone */
10293         ipm6.ipv6GroupMemberAddress = ilm->ilm_v6addr;
10294         ipm6.ipv6GroupMemberRefCnt = ilm->ilm_refcnt;
10295         ipm6.ipv6GroupMemberFilterMode = ilm->ilm_fmode;
10296         if (!snmp_append_data2(mpctl->b_cont,
10297             &mp_tail,
10298             (char *)&ipm6, (int)sizeof (ipm6))) {
10299             ipldbg(("ip_snmp_get_mib2_ip6_group: "
10300                 "failed to allocate %u bytes\n",
10301                 (uint_t)sizeof (ipm6)));
10302         }
10303     }
10304     rw_exit(&ill->ill_mcast_lock);
10305     ill_refrele(ill);
10306     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10307 }
10308 rw_exit(&ipst->ips_ill_g_lock);

10310 optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10311 ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10312     (int)optp->level, (int)optp->name, (int)optp->len));
10313 qreply(q, mpctl);
10314 return (mp2ctl);
10315 }

10317 /* IP multicast filtered sources */
10318 static mblk_t *
10319 ip_snmp_get_mib2_ip_group_src(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10320 {
10321     struct opthdr          *optp;
10322     mblk_t                 *mp2ctl;
10323     ill_t                 *ill;
10324     ipif_t               *ipif;
10325     ilm_t                 *ilm;
10326     ip_grpsrc_t          ips;
10327     mblk_t                 *mp_tail = NULL;
10328     ill_walk_context_t    ctx;
10329     zoneid_t             zoneid;
10330     int                   i;
10331     slist_t              *sl;

10333     /*
10334      * make a copy of the original message
10335      */
10336     mp2ctl = copymsg(mpctl);
10337     zoneid = Q_TO_CONN(q)->conn_zoneid;

10339     /* ipGroupSource table */
10340     optp = (struct opthdr *)&mpctl->b_rptr[
10341         sizeof (struct T_optmgmt_ack)];
10342     optp->level = MIB2_IP;
10343     optp->name = EXPER_IP_GROUP_SOURCES;

10345     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10346     ill = ILL_START_WALK_V4(&ctx, ipst);
10347     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10348         /* Make sure the ill isn't going away. */
10349         if (!ill_check_and_refhold(ill))
10350             continue;
10351         rw_exit(&ipst->ips_ill_g_lock);
10352         rw_enter(&ill->ill_mcast_lock, RW_READER);
10353         for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10354             sl = ilm->ilm_filter;
10355             if (ilm->ilm_zoneid != zoneid &&
10356                 ilm->ilm_zoneid != ALL_ZONES)
10357                 continue;

```

```

10358         if (SLIST_IS_EMPTY(sl))
10359             continue;

10361         /* Is there an ipif for ilm_ifaddr? */
10362         for (ipif = ill->ill_ipif; ipif != NULL;
10363             ipif = ipif->ipif_next) {
10364             if (!IPIF_IS_CONDEMNED(ipif) &&
10365                 ipif->ipif_lcl_addr == ilm->ilm_ifaddr &&
10366                 ilm->ilm_ifaddr != INADDR_ANY)
10367                 break;
10368         }
10369         if (ipif != NULL) {
10370             ipif_get_name(ipif,
10371                 ips.ipGroupSourceIfIndex.o_bytes,
10372                 OCTET_LENGTH);
10373         } else {
10374             ill_get_name(ill,
10375                 ips.ipGroupSourceIfIndex.o_bytes,
10376                 OCTET_LENGTH);
10377         }
10378         ips.ipGroupSourceIfIndex.o_length =
10379             mi_strlen(ips.ipGroupSourceIfIndex.o_bytes);

10381         ips.ipGroupSourceGroup = ilm->ilm_addr;
10382         for (i = 0; i < sl->sl_numsrc; i++) {
10383             if (!ING_IS_ADDR_V4MAPPED(&sl->sl_addr[i]))
10384                 continue;
10385             IN6_V4MAPPED_TO_IPADDR(&sl->sl_addr[i],
10386                 ips.ipGroupSourceAddress);
10387             if (snmp_append_data2(mpctl->b_cont, &mp_tail,
10388                 (char *)&ips, (int)sizeof (ips)) == 0) {
10389                 ipldbg(("ip_snmp_get_mib2_ip_group_src: "
10390                     " failed to allocate %u bytes\n",
10391                     (uint_t)sizeof (ips)));
10392             }
10393         }
10394     }
10395     rw_exit(&ill->ill_mcast_lock);
10396     ill_refrele(ill);
10397     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10398 }
10399 rw_exit(&ipst->ips_ill_g_lock);
10400 optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10401 ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10402     (int)optp->level, (int)optp->name, (int)optp->len));
10403 qreply(q, mpctl);
10404 return (mp2ctl);
10405 }

10407 /* IPv6 multicast filtered sources. */
10408 static mblk_t *
10409 ip_snmp_get_mib2_ip6_group_src(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10410 {
10411     struct opthdr          *optp;
10412     mblk_t                 *mp2ctl;
10413     ill_t                 *ill;
10414     ilm_t                 *ilm;
10415     ipv6_grpsrc_t        ips6;
10416     mblk_t                 *mp_tail = NULL;
10417     ill_walk_context_t    ctx;
10418     zoneid_t             zoneid;
10419     int                   i;
10420     slist_t              *sl;

10422     /*
10423      * make a copy of the original message

```

```

10424      */
10425      mp2ctl = copymsg(mpctl);
10426      zoneid = Q_TO_CONN(q)->conn_zoneid;

10428      /* ip6GroupMember table */
10429      optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10430      optp->level = MIB2_IP6;
10431      optp->name = EXPER_IP6_GROUP_SOURCES;

10433      rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10434      ill = ILL_START_WALK_V6(&ctx, ipst);
10435      for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10436          /* Make sure the ill isn't going away. */
10437          if (!ill_check_and_refhold(ill))
10438              continue;
10439          rw_exit(&ipst->ips_ill_g_lock);
10440          /*
10441           * Normally we don't have any members on under IPMP interfaces.
10442           * We report them as a debugging aid.
10443           */
10444          rw_enter(&ill->ill_mcast_lock, RW_READER);
10445          ips6.ipv6GroupSourceIfIndex = ill->ill_phyint->phyint_ifindex;
10446          for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10447              sl = ilm->ilm_filter;
10448              if (ilm->ilm_zoneid != zoneid &&
10449                  ilm->ilm_zoneid != ALL_ZONES)
10450                  continue;
10451              if (SLIST_IS_EMPTY(sl))
10452                  continue;
10453              ips6.ipv6GroupSourceGroup = ilm->ilm_v6addr;
10454              for (i = 0; i < sl->sl_numsrc; i++) {
10455                  ips6.ipv6GroupSourceAddress = sl->sl_addr[i];
10456                  if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10457                      (char *)&ips6, (int)sizeof(ips6))) {
10458                      ipldbg(("ip_snmp_get_mib2_ip6_"
10459                          "group_src: failed to allocate "
10460                          "%u bytes\n",
10461                          (uint_t)sizeof(ips6)));
10462                  }
10463              }
10464          }
10465          rw_exit(&ill->ill_mcast_lock);
10466          ill_refrele(ill);
10467          rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10468      }
10469      rw_exit(&ipst->ips_ill_g_lock);

10471      optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
10472      ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10473          (int)optp->level, (int)optp->name, (int)optp->len));
10474      greply(q, mpctl);
10475      return (mp2ctl);
10476 }

10478 /* Multicast routing virtual interface table. */
10479 static mblk_t *
10480 ip_snmp_get_mib2_virt_multi(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10481 {
10482     struct ophdr      *optp;
10483     mblk_t            *mp2ctl;

10485     /*
10486      * make a copy of the original message
10487      */
10488     mp2ctl = copymsg(mpctl);

```

```

10490     optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10491     optp->level = EXPER_DVMRP;
10492     optp->name = EXPER_DVMRP_VIF;
10493     if (!ip_mroute_vif(mpctl->b_cont, ipst)) {
10494         ip0dbg(("ip_mroute_vif: failed\n"));
10495     }
10496     optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
10497     ip3dbg(("ip_snmp_get_mib2_virt_multi: level %d, name %d, len %d\n",
10498         (int)optp->level, (int)optp->name, (int)optp->len));
10499     greply(q, mpctl);
10500     return (mp2ctl);
10501 }

10503 /* Multicast routing table. */
10504 static mblk_t *
10505 ip_snmp_get_mib2_multi_rtable(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10506 {
10507     struct ophdr      *optp;
10508     mblk_t            *mp2ctl;

10510     /*
10511      * make a copy of the original message
10512      */
10513     mp2ctl = copymsg(mpctl);

10515     optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10516     optp->level = EXPER_DVMRP;
10517     optp->name = EXPER_DVMRP_MRT;
10518     if (!ip_mroute_mrt(mpctl->b_cont, ipst)) {
10519         ip0dbg(("ip_mroute_mrt: failed\n"));
10520     }
10521     optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
10522     ip3dbg(("ip_snmp_get_mib2_multi_rtable: level %d, name %d, len %d\n",
10523         (int)optp->level, (int)optp->name, (int)optp->len));
10524     greply(q, mpctl);
10525     return (mp2ctl);
10526 }

10528 /*
10529  * Return ipRouteEntryTable, ipNetToMediaEntryTable, and ipRouteAttributeTable
10530  * in one IRE walk.
10531  */
10532 static mblk_t *
10533 ip_snmp_get_mib2_ip_route_media(queue_t *q, mblk_t *mpctl, int level,
10534     ip_stack_t *ipst)
10535 {
10536     struct ophdr      *optp;
10537     mblk_t            *mp2ctl;          /* Returned */
10538     mblk_t            *mp3ctl;          /* nettomedia */
10539     mblk_t            *mp4ctl;          /* routeattrs */
10540     ip_routedata_t    ird;
10541     zoneid_t          zoneid;

10543     /*
10544      * make copies of the original message
10545      * - mp2ctl is returned unchanged to the caller for his use
10546      * - mpctl is sent upstream as ipRouteEntryTable
10547      * - mp3ctl is sent upstream as ipNetToMediaEntryTable
10548      * - mp4ctl is sent upstream as ipRouteAttributeTable
10549      */
10550     mp2ctl = copymsg(mpctl);
10551     mp3ctl = copymsg(mpctl);
10552     mp4ctl = copymsg(mpctl);
10553     if (mp3ctl == NULL || mp4ctl == NULL) {
10554         freemsg(mp4ctl);
10555         freemsg(mp3ctl);

```

```

10556     freemsg(mp2ctl);
10557     freemsg(mpctl);
10558     return (NULL);
10559 }

10561 bzero(&ird, sizeof (ird));

10563 ird.ird_route.lp_head = mpctl->b_cont;
10564 ird.ird_netmedia.lp_head = mp3ctl->b_cont;
10565 ird.ird_attrs.lp_head = mp4ctl->b_cont;
10566 /*
10567  * If the level has been set the special EXPER_IP_AND_ALL_IRES value,
10568  * then also include ire_testhidden IRESs and IRE_IF_CLONE. This is
10569  * intended a temporary solution until a proper MIB API is provided
10570  * that provides complete filtering/caller-opt-in.
10571  */
10572 if (level == EXPER_IP_AND_ALL_IRES)
10573     ird.ird_flags |= IRD_REPORT_ALL;

10575 zoneid = Q_TO_CONN(q)->conn_zoneid;
10576 ire_walk_v4(ip_snmp_get2_v4, &ird, zoneid, ipst);

10578 /* ipRouteEntryTable in mpctl */
10579 optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10580 optp->level = MIB2_IP;
10581 optp->name = MIB2_IP_ROUTE;
10582 optp->len = msgdsize(ird.ird_route.lp_head);
10583 ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10584 (int)optp->level, (int)optp->name, (int)optp->len));
10585 qreply(q, mpctl);

10587 /* ipNetToMediaEntryTable in mp3ctl */
10588 ncec_walk(NULL, ip_snmp_get2_v4_media, &ird, ipst);

10590 optp = (struct ophdr *)&mp3ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10591 optp->level = MIB2_IP;
10592 optp->name = MIB2_IP_MEDIA;
10593 optp->len = msgdsize(ird.ird_netmedia.lp_head);
10594 ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10595 (int)optp->level, (int)optp->name, (int)optp->len));
10596 qreply(q, mp3ctl);

10598 /* ipRouteAttributeTable in mp4ctl */
10599 optp = (struct ophdr *)&mp4ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10600 optp->level = MIB2_IP;
10601 optp->name = EXPER_IP_RTATTR;
10602 optp->len = msgdsize(ird.ird_attrs.lp_head);
10603 ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10604 (int)optp->level, (int)optp->name, (int)optp->len));
10605 if (optp->len == 0)
10606     freemsg(mp4ctl);
10607 else
10608     qreply(q, mp4ctl);

10610 return (mp2ctl);
10611 }

10613 /*
10614  * Return ipv6RouteEntryTable and ipv6RouteAttributeTable in one IRE walk, and
10615  * ipv6NetToMediaEntryTable in an NDP walk.
10616  */
10617 static mblk_t *
10618 ip_snmp_get_mib2_ip6_route_media(queue_t *q, mblk_t *mpctl, int level,
10619 ip_stack_t *ipst)
10620 {
10621     struct ophdr *optp;

```

```

10622     mblk_t *mp2ctl; /* Returned */
10623     mblk_t *mp3ctl; /* nettomedia */
10624     mblk_t *mp4ctl; /* routeattrs */
10625     iproutedata_t ird;
10626     zoneid_t zoneid;

10628 /*
10629  * make copies of the original message
10630  * - mp2ctl is returned unchanged to the caller for his use
10631  * - mpctl is sent upstream as ipv6RouteEntryTable
10632  * - mp3ctl is sent upstream as ipv6NetToMediaEntryTable
10633  * - mp4ctl is sent upstream as ipv6RouteAttributeTable
10634  */
10635 mp2ctl = copymsg(mpctl);
10636 mp3ctl = copymsg(mpctl);
10637 mp4ctl = copymsg(mpctl);
10638 if (mp3ctl == NULL || mp4ctl == NULL) {
10639     freemsg(mp4ctl);
10640     freemsg(mp3ctl);
10641     freemsg(mp2ctl);
10642     freemsg(mpctl);
10643     return (NULL);
10644 }

10646 bzero(&ird, sizeof (ird));

10648 ird.ird_route.lp_head = mpctl->b_cont;
10649 ird.ird_netmedia.lp_head = mp3ctl->b_cont;
10650 ird.ird_attrs.lp_head = mp4ctl->b_cont;
10651 /*
10652  * If the level has been set the special EXPER_IP_AND_ALL_IRES value,
10653  * then also include ire_testhidden IRESs and IRE_IF_CLONE. This is
10654  * intended a temporary solution until a proper MIB API is provided
10655  * that provides complete filtering/caller-opt-in.
10656  */
10657 if (level == EXPER_IP_AND_ALL_IRES)
10658     ird.ird_flags |= IRD_REPORT_ALL;

10660 zoneid = Q_TO_CONN(q)->conn_zoneid;
10661 ire_walk_v6(ip_snmp_get2_v6_route, &ird, zoneid, ipst);

10663 optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10664 optp->level = MIB2_IP6;
10665 optp->name = MIB2_IP6_ROUTE;
10666 optp->len = msgdsize(ird.ird_route.lp_head);
10667 ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10668 (int)optp->level, (int)optp->name, (int)optp->len));
10669 qreply(q, mpctl);

10671 /* ipv6NetToMediaEntryTable in mp3ctl */
10672 ncec_walk(NULL, ip_snmp_get2_v6_media, &ird, ipst);

10674 optp = (struct ophdr *)&mp3ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10675 optp->level = MIB2_IP6;
10676 optp->name = MIB2_IP6_MEDIA;
10677 optp->len = msgdsize(ird.ird_netmedia.lp_head);
10678 ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10679 (int)optp->level, (int)optp->name, (int)optp->len));
10680 qreply(q, mp3ctl);

10682 /* ipv6RouteAttributeTable in mp4ctl */
10683 optp = (struct ophdr *)&mp4ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10684 optp->level = MIB2_IP6;
10685 optp->name = EXPER_IP_RTATTR;
10686 optp->len = msgdsize(ird.ird_attrs.lp_head);
10687 ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",

```



```

10688     (int)optp->level, (int)optp->name, (int)optp->len));
10689     if (optp->len == 0)
10690         freemsg(mp4ctl);
10691     else
10692         qreply(q, mp4ctl);
10694     return (mp2ctl);
10695 }

10697 /*
10698  * IPv6 mib: One per ill
10699  */
10700 static mblk_t *
10701 ip_snmp_get_mib2_ip6(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
10702     boolean_t legacy_req)
10703 {
10704     struct opthdr      *optp;
10705     mblk_t             *mp2ctl;
10706     ill_t              *ill;
10707     ill_walk_context_t ctx;
10708     mblk_t             *mp_tail = NULL;
10709     mib2_ipv6AddrEntry_t mae6;
10710     mib2_ipIfStatsEntry_t *ise;
10711     size_t             ise_size, iae_size;

10713     /*
10714      * Make a copy of the original message
10715      */
10716     mp2ctl = copymsg(mpctl);

10718     /* fixed length IPv6 structure ... */

10720     if (legacy_req) {
10721         ise_size = LEGACY_MIB_SIZE(&ipst->ips_ip6_mib,
10722             mib2_ipIfStatsEntry_t);
10723         iae_size = LEGACY_MIB_SIZE(&mae6, mib2_ipv6AddrEntry_t);
10724     } else {
10725         ise_size = sizeof (mib2_ipIfStatsEntry_t);
10726         iae_size = sizeof (mib2_ipv6AddrEntry_t);
10727     }

10729     optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10730     optp->level = MIB2_IP6;
10731     optp->name = 0;
10732     /* Include "unknown interface" ip6_mib */
10733     ipst->ips_ip6_mib.ipIfStatsIPVersion = MIB2_INETADDRESSSTYPE_ipv6;
10734     ipst->ips_ip6_mib.ipIfStatsIfIndex =
10735         MIB2_UNKNOWN_INTERFACE; /* Flag to netstat */
10736     SET_MIB(ipst->ips_ip6_mib.ipIfStatsForwarding,
10737         ipst->ips_ipv6_forwarding ? 1 : 2);
10738     SET_MIB(ipst->ips_ip6_mib.ipIfStatsDefaultHopLimit,
10739         ipst->ips_ipv6_def_hops);
10740     SET_MIB(ipst->ips_ip6_mib.ipIfStatsEntrySize,
10741         sizeof (mib2_ipIfStatsEntry_t));
10742     SET_MIB(ipst->ips_ip6_mib.ipIfStatsAddrEntrySize,
10743         sizeof (mib2_ipv6AddrEntry_t));
10744     SET_MIB(ipst->ips_ip6_mib.ipIfStatsRouteEntrySize,
10745         sizeof (mib2_ipv6RouteEntry_t));
10746     SET_MIB(ipst->ips_ip6_mib.ipIfStatsNetToMediaEntrySize,
10747         sizeof (mib2_ipv6NetToMediaEntry_t));
10748     SET_MIB(ipst->ips_ip6_mib.ipIfStatsMemberEntrySize,
10749         sizeof (ipv6_member_t));
10750     SET_MIB(ipst->ips_ip6_mib.ipIfStatsGroupSourceEntrySize,
10751         sizeof (ipv6_grpsrc_t));

10753     /*

```

```

10754     * Synchronize 64- and 32-bit counters
10755     */
10756     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInReceives,
10757         ipIfStatsHCInReceives);
10758     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInDelivers,
10759         ipIfStatsHCInDelivers);
10760     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutRequests,
10761         ipIfStatsHCOutRequests);
10762     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutForwDatagrams,
10763         ipIfStatsHCOutForwDatagrams);
10764     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutMcastPkts,
10765         ipIfStatsHCOutMcastPkts);
10766     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInMcastPkts,
10767         ipIfStatsHCInMcastPkts);

10769     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10770         (char *)&ipst->ips_ip6_mib, (int)ise_size)) {
10771         ipldb("ip_snmp_get_mib2_ip6: failed to allocate %u bytes\n",
10772             (uint_t)ise_size);
10773     } else if (legacy_req) {
10774         /* Adjust the EntrySize fields for legacy requests. */
10775         ise =
10776             (mib2_ipIfStatsEntry_t *) (mp_tail->b_wptr - (int)ise_size);
10777         SET_MIB(ise->ipIfStatsEntrySize, ise_size);
10778         SET_MIB(ise->ipIfStatsAddrEntrySize, iae_size);
10779     }

10781     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10782     ill = ILL_START_WALK_V6(&ctx, ipst);
10783     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10784         ill->ill_ip_mib->ipIfStatsIfIndex =
10785             ill->ill_phyint->phyint_ifindex;
10786         SET_MIB(ill->ill_ip_mib->ipIfStatsForwarding,
10787             ipst->ips_ipv6_forwarding ? 1 : 2);
10788         SET_MIB(ill->ill_ip_mib->ipIfStatsDefaultHopLimit,
10789             ill->ill_max_hops);

10791     /*
10792      * Synchronize 64- and 32-bit counters
10793      */
10794     SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInReceives,
10795         ipIfStatsHCInReceives);
10796     SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInDelivers,
10797         ipIfStatsHCInDelivers);
10798     SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutRequests,
10799         ipIfStatsHCOutRequests);
10800     SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutForwDatagrams,
10801         ipIfStatsHCOutForwDatagrams);
10802     SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutMcastPkts,
10803         ipIfStatsHCOutMcastPkts);
10804     SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInMcastPkts,
10805         ipIfStatsHCInMcastPkts);

10807     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10808         (char *)ill->ill_ip_mib, (int)ise_size)) {
10809         ipldb("ip_snmp_get_mib2_ip6: failed to allocate "
10810             "%u bytes\n", (uint_t)ise_size);
10811     } else if (legacy_req) {
10812         /* Adjust the EntrySize fields for legacy requests. */
10813         ise = (mib2_ipIfStatsEntry_t *) (mp_tail->b_wptr -
10814             (int)ise_size);
10815         SET_MIB(ise->ipIfStatsEntrySize, ise_size);
10816         SET_MIB(ise->ipIfStatsAddrEntrySize, iae_size);
10817     }
10818     }
10819     rw_exit(&ipst->ips_ill_g_lock);

```

```

10821     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10822     ip3dbg(("ip_snmp_get_mib2 ip6: level %d, name %d, len %d\n",
10823         (int)optp->level, (int)optp->name, (int)optp->len));
10824     qreply(q, mpctl);
10825     return (mp2ctl);
10826 }

10828 /*
10829  * ICMPv6 mib: One per ill
10830  */
10831 static mblk_t *
10832 ip_snmp_get_mib2_icmp6(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10833 {
10834     struct ophdr      *optp;
10835     mblk_t            *mp2ctl;
10836     ill_t             *ill;
10837     ill_walk_context_t ctx;
10838     mblk_t            *mp_tail = NULL;
10839     /*
10840      * Make a copy of the original message
10841      */
10842     mp2ctl = copymsg(mpctl);

10844     /* fixed length ICMPv6 structure ... */

10846     optp = (struct ophdr *)&mpctl->b_rptr[ sizeof (struct T_optmgmt_ack)];
10847     optp->level = MIB2_ICMP6;
10848     optp->name = 0;
10849     /* Include "unknown interface" icmp6_mib */
10850     ipst->ips_icmp6_mib.ipv6IfIcmpIfIndex =
10851         MIB2_UNKNOWN_INTERFACE; /* netstat flag */
10852     ipst->ips_icmp6_mib.ipv6IfIcmpEntrySize =
10853         sizeof (mib2_ipv6IfIcmpEntry_t);
10854     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10855         (char *)&ipst->ips_icmp6_mib,
10856         (int)sizeof (ipst->ips_icmp6_mib))) {
10857         ip1dbg(("ip_snmp_get_mib2_icmp6: failed to allocate %u bytes\n",
10858             (uint_t)sizeof (ipst->ips_icmp6_mib)));
10859     }

10861     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10862     ill = ILL_START_WALK_V6(&ctx, ipst);
10863     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10864         ill->ill_icmp6_mib->ipv6IfIcmpIfIndex =
10865             ill->ill_phyint->phyint_ifindex;
10866         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10867             (char *)ill->ill_icmp6_mib,
10868             (int)sizeof (*ill->ill_icmp6_mib))) {
10869             ip1dbg(("ip_snmp_get_mib2_icmp6: failed to allocate "
10870                 "%u bytes\n",
10871                 (uint_t)sizeof (*ill->ill_icmp6_mib)));
10872         }
10873     }
10874     rw_exit(&ipst->ips_ill_g_lock);

10876     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10877     ip3dbg(("ip_snmp_get_mib2_icmp6: level %d, name %d, len %d\n",
10878         (int)optp->level, (int)optp->name, (int)optp->len));
10879     qreply(q, mpctl);
10880     return (mp2ctl);
10881 }

10883 /*
10884  * ire_walk routine to create both ipRouteEntryTable and
10885  * ipRouteAttributeTable in one IRE walk

```

```

10886  */
10887 static void
10888 ip_snmp_get2_v4(ire_t *ire, iproutedata_t *ird)
10889 {
10890     ill_t             *ill;
10891     mib2_ipRouteEntry_t *re;
10892     mib2_ipAttributeEntry_t *aes;
10893     tsol_ire_gw_secattr_t *attrp;
10894     tsol_gc_t         *gc = NULL;
10895     tsol_gcgrp_t     *gcgrp = NULL;
10896     ip_stack_t       *ipst = ire->ire_ipst;

10898     ASSERT(ire->ire_ipversion == IPV4_VERSION);

10900     if (!(ird->ird_flags & IRD_REPORT_ALL)) {
10901         if (ire->ire_testhidden)
10902             return;
10903         if (ire->ire_type & IRE_IF_CLONE)
10904             return;
10905     }

10907     if ((re = kmem_zalloc(sizeof (*re), KM_NOSLEEP)) == NULL)
10908         return;

10910     if ((attrp = ire->ire_gw_secattr) != NULL) {
10911         mutex_enter(&attrp->igsa_lock);
10912         if ((gc = attrp->igsa_gc) != NULL) {
10913             gcgrp = gc->gc_grp;
10914             ASSERT(gcgrp != NULL);
10915             rw_enter(&gcgrp->gcgrp_rwlock, RW_READER);
10916         }
10917         mutex_exit(&attrp->igsa_lock);
10918     }
10919     /*
10920      * Return all IRE types for route table... let caller pick and choose
10921      */
10922     re->ipRouteDest = ire->ire_addr;
10923     ill = ire->ire_ill;
10924     re->ipRouteIfIndex.o_length = 0;
10925     if (ill != NULL) {
10926         ill_get_name(ill, re->ipRouteIfIndex.o_bytes, OCTET_LENGTH);
10927         re->ipRouteIfIndex.o_length =
10928             mi_strlen(re->ipRouteIfIndex.o_bytes);
10929     }
10930     re->ipRouteMetric1 = -1;
10931     re->ipRouteMetric2 = -1;
10932     re->ipRouteMetric3 = -1;
10933     re->ipRouteMetric4 = -1;

10935     re->ipRouteNextHop = ire->ire_gateway_addr;
10936     /* indirect(4), direct(3), or invalid(2) */
10937     if (ire->ire_flags & (RTF_REJECT | RTF_BLACKHOLE))
10938         re->ipRouteType = 2;
10939     else if (ire->ire_type & IRE_ONLINK)
10940         re->ipRouteType = 3;
10941     else
10942         re->ipRouteType = 4;

10944     re->ipRouteProto = -1;
10945     re->ipRouteAge = gethrestime_sec() - ire->ire_create_time;
10946     re->ipRouteMask = ire->ire_mask;
10947     re->ipRouteMetric5 = -1;
10948     re->ipRouteInfo.re_max_frag = ire->ire_metrics.iulp_mtu;
10949     if (ire->ire_ill != NULL && re->ipRouteInfo.re_max_frag == 0)
10950         re->ipRouteInfo.re_max_frag = ire->ire_ill->ill_mtu;

```

```

10952     re->ipRouteInfo.re_frag_flag = 0;
10953     re->ipRouteInfo.re_rtt        = 0;
10954     re->ipRouteInfo.re_src_addr  = 0;
10955     re->ipRouteInfo.re_ref       = ire->ire_refcnt;
10956     re->ipRouteInfo.re_obpkt     = ire->ire_ob_pkt_count;
10957     re->ipRouteInfo.re_ibpkt     = ire->ire_ib_pkt_count;
10958     re->ipRouteInfo.re_flags     = ire->ire_flags;

10960     /* Add the IRE_IF_CLONE's counters to their parent IRE_INTERFACE */
10961     if (ire->ire_type & IRE_INTERFACE) {
10962         ire_t *child;

10964         rw_enter(&ipst->ips_ire_dep_lock, RW_READER);
10965         child = ire->ire_dep_children;
10966         while (child != NULL) {
10967             re->ipRouteInfo.re_obpkt += child->ire_ob_pkt_count;
10968             re->ipRouteInfo.re_ibpkt += child->ire_ib_pkt_count;
10969             child = child->ire_dep_sib_next;
10970         }
10971         rw_exit(&ipst->ips_ire_dep_lock);
10972     }

10974     if (ire->ire_flags & RTF_DYNAMIC) {
10975         re->ipRouteInfo.re_ire_type = IRE_HOST_REDIRECT;
10976     } else {
10977         re->ipRouteInfo.re_ire_type = ire->ire_type;
10978     }

10980     if (!snmp_append_data2(ird->ird_route.lp_head, &ird->ird_route.lp_tail,
10981         (char *)re, (int)sizeof (*re))) {
10982         ipldbg(("ip_snmp_get2_v4: failed to allocate %u bytes\n",
10983             (uint_t)sizeof (*re)));
10984     }

10986     if (gc != NULL) {
10987         iaes.iae_routeidx = ird->ird_idx;
10988         iaes.iae_doi = gc->gc_db->gcdb_doi;
10989         iaes.iae_slrange = gc->gc_db->gcdb_slrange;

10991         if (!snmp_append_data2(ird->ird_attrs.lp_head,
10992             &ird->ird_attrs.lp_tail, (char *)&iaes, sizeof (iaes))) {
10993             ipldbg(("ip_snmp_get2_v4: failed to allocate %u "
10994                 "bytes\n", (uint_t)sizeof (iaes)));
10995         }
10996     }

10998     /* bump route index for next pass */
10999     ird->ird_idx++;

11001     kmem_free(re, sizeof (*re));
11002     if (gcgrp != NULL)
11003         rw_exit(&gcgrp->gcgrp_rwlock);
11004 }

11006 /*
11007  * ire_walk routine to create ipv6RouteEntryTable and ipRouteEntryTable.
11008  */
11009 static void
11010 ip_snmp_get2_v6_route(ire_t *ire, iproutedata_t *ird)
11011 {
11012     ill_t *ill;
11013     mib2_ipv6RouteEntry_t *re;
11014     mib2_ipAttributeEntry_t iaes;
11015     tsol_ire_gw_secattr_t *attrp;
11016     tsol_gc_t *gc = NULL;
11017     tsol_gcgrp_t *gcgrp = NULL;

```

```

11018     ip_stack_t *ipst = ire->ire_ipst;

11020     ASSERT(ire->ire_ipversion == IPV6_VERSION);

11022     if (!(ird->ird_flags & IRD_REPORT_ALL)) {
11023         if (ire->ire_testhidden)
11024             return;
11025         if (ire->ire_type & IRE_IF_CLONE)
11026             return;
11027     }

11029     if ((re = kmem_zalloc(sizeof (*re), KM_NOSLEEP)) == NULL)
11030         return;

11032     if ((attrp = ire->ire_gw_secattr) != NULL) {
11033         mutex_enter(&attrp->igsa_lock);
11034         if ((gc = attrp->igsa_gc) != NULL) {
11035             gcgrp = gc->gc_grp;
11036             ASSERT(gcgrp != NULL);
11037             rw_enter(&gcgrp->gcgrp_rwlock, RW_READER);
11038         }
11039         mutex_exit(&attrp->igsa_lock);
11040     }
11041     /*
11042      * Return all IRE types for route table... let caller pick and choose
11043      */
11044     re->ipv6RouteDest = ire->ire_addr_v6;
11045     re->ipv6RoutePfxLength = ip_mask_to_plen_v6(&ire->ire_mask_v6);
11046     re->ipv6RouteIndex = 0; /* Unique when multiple with same dest/plen */
11047     re->ipv6RouteIfIndex.o_length = 0;
11048     ill = ire->ire_ill;
11049     if (ill != NULL) {
11050         ill_get_name(ill, re->ipv6RouteIfIndex.o_bytes, OCTET_LENGTH);
11051         re->ipv6RouteIfIndex.o_length =
11052             mi_strlen(re->ipv6RouteIfIndex.o_bytes);
11053     }

11055     ASSERT(!(ire->ire_type & IRE_BROADCAST));

11057     mutex_enter(&ire->ire_lock);
11058     re->ipv6RouteNextHop = ire->ire_gateway_addr_v6;
11059     mutex_exit(&ire->ire_lock);

11061     /* remote(4), local(3), or discard(2) */
11062     if (ire->ire_flags & (RTF_REJECT | RTF_BLACKHOLE))
11063         re->ipv6RouteType = 2;
11064     else if (ire->ire_type & IRE_ONLINK)
11065         re->ipv6RouteType = 3;
11066     else
11067         re->ipv6RouteType = 4;

11069     re->ipv6RouteProtocol = -1;
11070     re->ipv6RoutePolicy = 0;
11071     re->ipv6RouteAge = gethrestime_sec() - ire->ire_create_time;
11072     re->ipv6RouteNextHopRDI = 0;
11073     re->ipv6RouteWeight = 0;
11074     re->ipv6RouteMetric = 0;
11075     re->ipv6RouteInfo.re_max_frag = ire->ire_metrics.iulp_mtu;
11076     if (ire->ire_ill != NULL && re->ipv6RouteInfo.re_max_frag == 0)
11077         re->ipv6RouteInfo.re_max_frag = ire->ire_ill->ill_mtu;

11079     re->ipv6RouteInfo.re_frag_flag = 0;
11080     re->ipv6RouteInfo.re_rtt = 0;
11081     re->ipv6RouteInfo.re_src_addr = ipv6_all_zeros;
11082     re->ipv6RouteInfo.re_obpkt = ire->ire_ob_pkt_count;
11083     re->ipv6RouteInfo.re_ibpkt = ire->ire_ib_pkt_count;

```

```

11084 re->ipv6RouteInfo.re_ref = ire->ire_refcnt;
11085 re->ipv6RouteInfo.re_flags = ire->ire_flags;

11087 /* Add the IRE_IF_CLONE's counters to their parent IRE_INTERFACE */
11088 if (ire->ire_type & IRE_INTERFACE) {
11089     ire_t *child;

11091     rw_enter(&ipst->ips_ire_dep_lock, RW_READER);
11092     child = ire->ire_dep_children;
11093     while (child != NULL) {
11094         re->ipv6RouteInfo.re_obpkt += child->ire_ob_pkt_count;
11095         re->ipv6RouteInfo.re_ibpkt += child->ire_ib_pkt_count;
11096         child = child->ire_dep_sib_next;
11097     }
11098     rw_exit(&ipst->ips_ire_dep_lock);
11099 }
11100 if (ire->ire_flags & RTE_DYNAMIC) {
11101     re->ipv6RouteInfo.re_ire_type = IRE_HOST_REDIRECT;
11102 } else {
11103     re->ipv6RouteInfo.re_ire_type = ire->ire_type;
11104 }

11106 if (!snmp_append_data2(ird->ird_route.lp_head, &ird->ird_route.lp_tail,
11107 (char *)re, (int)sizeof (*re))) {
11108     ipldbg(("ip_snmp_get2_v6: failed to allocate %u bytes\n",
11109 (uint_t)sizeof (*re)));
11110 }

11112 if (gc != NULL) {
11113     iaes.iae_routeidx = ird->ird_idx;
11114     iaes.iae_doi = gc->gc_db->gcdb_doi;
11115     iaes.iae_slrange = gc->gc_db->gcdb_slrange;

11117     if (!snmp_append_data2(ird->ird_attrs.lp_head,
11118 &ird->ird_attrs.lp_tail, (char *)&iaes, sizeof (iaes))) {
11119         ipldbg(("ip_snmp_get2_v6: failed to allocate %u "
11120 "bytes\n", (uint_t)sizeof (iaes)));
11121     }
11122 }

11124 /* bump route index for next pass */
11125 ird->ird_idx++;

11127 kmem_free(re, sizeof (*re));
11128 if (gcgrp != NULL)
11129     rw_exit(&gcgrp->gcgrp_rwlock);
11130 }

11132 /*
11133  * ncec_walk routine to create ipv6NetToMediaEntryTable
11134  */
11135 static int
11136 ip_snmp_get2_v6_media(nccec_t *ncec, iproutedata_t *ird)
11137 {
11138     ill_t *ill;
11139     mib2_ipv6NetToMediaEntry_t ntme;

11141     ill = ncec->ncec_ill;
11142     /* skip arpce entries, and loopback ncec entries */
11143     if (ill->ill_isv6 == B_FALSE || ill->ill_net_type == IRE_LOOPBACK)
11144         return (0);
11145     /*
11146      * Neighbor cache entry attached to IRE with on-link
11147      * destination.
11148      * We report all IPMP groups on ncec_ill which is normally the upper.
11149      */

```

```

11150     ntme.ipv6NetToMediaIfIndex = ill->ill_phyint->phyint_ifindex;
11151     ntme.ipv6NetToMediaNetAddress = ncec->ncec_addr;
11152     ntme.ipv6NetToMediaPhysAddress.o_length = ill->ill_phys_addr_length;
11153     if (ncec->ncec_lladdr != NULL) {
11154         bcopy(ncec->ncec_lladdr, ntme.ipv6NetToMediaPhysAddress.o_bytes,
11155             ntme.ipv6NetToMediaPhysAddress.o_length);
11156     }
11157     /*
11158      * Note: Returns ND_* states. Should be:
11159      * reachable(1), stale(2), delay(3), probe(4),
11160      * invalid(5), unknown(6)
11161      */
11162     ntme.ipv6NetToMediaState = ncec->ncec_state;
11163     ntme.ipv6NetToMediaLastUpdated = 0;

11165     /* other(1), dynamic(2), static(3), local(4) */
11166     if (NCE_MYADDR(ncec)) {
11167         ntme.ipv6NetToMediaType = 4;
11168     } else if (ncec->ncec_flags & NCE_F_PUBLISH) {
11169         ntme.ipv6NetToMediaType = 1; /* proxy */
11170     } else if (ncec->ncec_flags & NCE_F_STATIC) {
11171         ntme.ipv6NetToMediaType = 3;
11172     } else if (ncec->ncec_flags & (NCE_F_MCAST|NCE_F_BCAST)) {
11173         ntme.ipv6NetToMediaType = 1;
11174     } else {
11175         ntme.ipv6NetToMediaType = 2;
11176     }

11178     if (!snmp_append_data2(ird->ird_netmedia.lp_head,
11179 &ird->ird_netmedia.lp_tail, (char *)&ntme, sizeof (ntme))) {
11180         ipldbg(("ip_snmp_get2_v6_media: failed to allocate %u bytes\n",
11181 (uint_t)sizeof (ntme)));
11182     }
11183     return (0);
11184 }

11186 int
11187 ncec2ace(nccec_t *ncec)
11188 {
11189     int flags = 0;

11191     if (NCE_ISREACHABLE(ncec))
11192         flags |= ACE_F_RESOLVED;
11193     if (ncec->ncec_flags & NCE_F_AUTHORITY)
11194         flags |= ACE_F_AUTHORITY;
11195     if (ncec->ncec_flags & NCE_F_PUBLISH)
11196         flags |= ACE_F_PUBLISH;
11197     if ((ncec->ncec_flags & NCE_F_NONUD) != 0)
11198         flags |= ACE_F_PERMANENT;
11199     if (NCE_MYADDR(ncec))
11200         flags |= (ACE_F_MYADDR | ACE_F_AUTHORITY);
11201     if (ncec->ncec_flags & NCE_F_UNVERIFIED)
11202         flags |= ACE_F_UNVERIFIED;
11203     if (ncec->ncec_flags & NCE_F_AUTHORITY)
11204         flags |= ACE_F_AUTHORITY;
11205     if (ncec->ncec_flags & NCE_F_DELAYED)
11206         flags |= ACE_F_DELAYED;
11207     return (flags);
11208 }

11210 /*
11211  * ncec_walk routine to create ipNetToMediaEntryTable
11212  */
11213 static int
11214 ip_snmp_get2_v4_media(nccec_t *ncec, iproutedata_t *ird)
11215 {

```

```

11216     ill_t                *ill;
11217     mib2_ipNetToMediaEntry_t ntm;
11218     const char            *name = "unknown";
11219     ipaddr_t             ncec_addr;

11221     ill = ncec->ncec_ill;
11222     if (ill->ill_isv6 || (ncec->ncec_flags & NCE_F_BCAST) ||
11223         ill->ill_net_type == IRE_LOOPBACK)
11224         return (0);

11226     /* We report all IPMP groups on ncec_ill which is normally the upper. */
11227     name = ill->ill_name;
11228     /* Based on RFC 4293: other(1), inval(2), dyn(3), stat(4) */
11229     if (NCE_MYADDR(ncec)) {
11230         ntm.ipNetToMediaType = 4;
11231     } else if (ncec->ncec_flags & (NCE_F_MCAST|NCE_F_BCAST|NCE_F_PUBLISH)) {
11232         ntm.ipNetToMediaType = 1;
11233     } else {
11234         ntm.ipNetToMediaType = 3;
11235     }
11236     ntm.ipNetToMediaIfIndex.o_length = MIN(OCTET_LENGTH, strlen(name));
11237     bcopy(name, ntm.ipNetToMediaIfIndex.o_bytes,
11238           ntm.ipNetToMediaIfIndex.o_length);

11240     IN6_V4MAPPED_TO_IPADDR(&ncec->ncec_addr, ncec_addr);
11241     bcopy(&ncec_addr, &ntm.ipNetToMediaNetAddress, sizeof(ncec_addr));

11243     ntm.ipNetToMediaInfo.ntm_mask.o_length = sizeof(ipaddr_t);
11244     ncec_addr = INADDR_BROADCAST;
11245     bcopy(&ncec_addr, ntm.ipNetToMediaInfo.ntm_mask.o_bytes,
11246           sizeof(ncec_addr));
11247     /*
11248      * map all the flags to the ACE counterpart.
11249      */
11250     ntm.ipNetToMediaInfo.ntm_flags = nce2ace(ncec);

11252     ntm.ipNetToMediaPhysAddress.o_length =
11253         MIN(OCTET_LENGTH, ill->ill_phys_addr_length);

11255     if (!NCE_ISREACHABLE(ncec))
11256         ntm.ipNetToMediaPhysAddress.o_length = 0;
11257     else {
11258         if (ncec->ncec_lladdr != NULL) {
11259             bcopy(ncec->ncec_lladdr,
11260                   ntm.ipNetToMediaPhysAddress.o_bytes,
11261                   ntm.ipNetToMediaPhysAddress.o_length);
11262         }
11263     }

11265     if (!snmp_append_data2(ird->ird_netmedia.lp_head,
11266                          &ird->ird_netmedia.lp_tail, (char *)&ntm, sizeof(ntm))) {
11267         ipldbg(("ip_snmp_get2_v4 media: failed to allocate %u bytes\n",
11268               (uint_t)sizeof(ntm)));
11269     }
11270     return (0);
11271 }

11273 /*
11274  * return (0) if invalid set request, 1 otherwise, including non-tcp requests
11275  */
11276 /* ARGSUSED */
11277 int
11278 ip_snmp_set(queue_t *q, int level, int name, uchar_t *ptr, int len)
11279 {
11280     switch (level) {
11281     case MIB2_IP:

```

```

11282     case MIB2_ICMP:
11283         switch (name) {
11284             default:
11285                 break;
11286         }
11287         return (1);
11288     default:
11289         return (1);
11290 }
11291 }

11293 /*
11294  * When there exists both a 64- and 32-bit counter of a particular type
11295  * (i.e., InReceives), only the 64-bit counters are added.
11296  */
11297 void
11298 ip_mib2_add_ip_stats(mib2_ipIfStatsEntry_t *o1, mib2_ipIfStatsEntry_t *o2)
11299 {
11300     UPDATE_MIB(o1, ipIfStatsInHdrErrors, o2->ipIfStatsInHdrErrors);
11301     UPDATE_MIB(o1, ipIfStatsInTooBigErrors, o2->ipIfStatsInTooBigErrors);
11302     UPDATE_MIB(o1, ipIfStatsInNoRoutes, o2->ipIfStatsInNoRoutes);
11303     UPDATE_MIB(o1, ipIfStatsInAddrErrors, o2->ipIfStatsInAddrErrors);
11304     UPDATE_MIB(o1, ipIfStatsInUnknownProtos, o2->ipIfStatsInUnknownProtos);
11305     UPDATE_MIB(o1, ipIfStatsInTruncatedPkts, o2->ipIfStatsInTruncatedPkts);
11306     UPDATE_MIB(o1, ipIfStatsInDiscards, o2->ipIfStatsInDiscards);
11307     UPDATE_MIB(o1, ipIfStatsOutDiscards, o2->ipIfStatsOutDiscards);
11308     UPDATE_MIB(o1, ipIfStatsOutFragOKs, o2->ipIfStatsOutFragOKs);
11309     UPDATE_MIB(o1, ipIfStatsOutFragFails, o2->ipIfStatsOutFragFails);
11310     UPDATE_MIB(o1, ipIfStatsOutFragCreates, o2->ipIfStatsOutFragCreates);
11311     UPDATE_MIB(o1, ipIfStatsReasmReqds, o2->ipIfStatsReasmReqds);
11312     UPDATE_MIB(o1, ipIfStatsReasmOKs, o2->ipIfStatsReasmOKs);
11313     UPDATE_MIB(o1, ipIfStatsReasmFails, o2->ipIfStatsReasmFails);
11314     UPDATE_MIB(o1, ipIfStatsOutNoRoutes, o2->ipIfStatsOutNoRoutes);
11315     UPDATE_MIB(o1, ipIfStatsReasmDuplicates, o2->ipIfStatsReasmDuplicates);
11316     UPDATE_MIB(o1, ipIfStatsReasmPartDups, o2->ipIfStatsReasmPartDups);
11317     UPDATE_MIB(o1, ipIfStatsForwProhibits, o2->ipIfStatsForwProhibits);
11318     UPDATE_MIB(o1, udpInCksumErrs, o2->udpInCksumErrs);
11319     UPDATE_MIB(o1, udpInOverflows, o2->udpInOverflows);
11320     UPDATE_MIB(o1, rawipInOverflows, o2->rawipInOverflows);
11321     UPDATE_MIB(o1, ipIfStatsInWrongIPVersion,
11322               o2->ipIfStatsInWrongIPVersion);
11323     UPDATE_MIB(o1, ipIfStatsOutWrongIPVersion,
11324               o2->ipIfStatsInWrongIPVersion);
11325     UPDATE_MIB(o1, ipIfStatsOutSwitchIPVersion,
11326               o2->ipIfStatsOutSwitchIPVersion);
11327     UPDATE_MIB(o1, ipIfStatsHCInReceives, o2->ipIfStatsHCInReceives);
11328     UPDATE_MIB(o1, ipIfStatsHCInOctets, o2->ipIfStatsHCInOctets);
11329     UPDATE_MIB(o1, ipIfStatsHCInForwDatagrams,
11330               o2->ipIfStatsHCInForwDatagrams);
11331     UPDATE_MIB(o1, ipIfStatsHCInDelivers, o2->ipIfStatsHCInDelivers);
11332     UPDATE_MIB(o1, ipIfStatsHCOutRequests, o2->ipIfStatsHCOutRequests);
11333     UPDATE_MIB(o1, ipIfStatsHCOutForwDatagrams,
11334               o2->ipIfStatsHCOutForwDatagrams);
11335     UPDATE_MIB(o1, ipIfStatsOutFragReqds, o2->ipIfStatsOutFragReqds);
11336     UPDATE_MIB(o1, ipIfStatsHCOutTransmits, o2->ipIfStatsHCOutTransmits);
11337     UPDATE_MIB(o1, ipIfStatsHCOutOctets, o2->ipIfStatsHCOutOctets);
11338     UPDATE_MIB(o1, ipIfStatsHCInMcastPkts, o2->ipIfStatsHCInMcastPkts);
11339     UPDATE_MIB(o1, ipIfStatsHCInMcastOctets, o2->ipIfStatsHCInMcastOctets);
11340     UPDATE_MIB(o1, ipIfStatsHCOutMcastPkts, o2->ipIfStatsHCOutMcastPkts);
11341     UPDATE_MIB(o1, ipIfStatsHCOutMcastOctets,
11342               o2->ipIfStatsHCOutMcastOctets);
11343     UPDATE_MIB(o1, ipIfStatsHCInBcastPkts, o2->ipIfStatsHCInBcastPkts);
11344     UPDATE_MIB(o1, ipIfStatsHCOutBcastPkts, o2->ipIfStatsHCOutBcastPkts);
11345     UPDATE_MIB(o1, ipsecInSucceeded, o2->ipsecInSucceeded);
11346     UPDATE_MIB(o1, ipsecInFailed, o2->ipsecInFailed);
11347     UPDATE_MIB(o1, ipInCksumErrs, o2->ipInCksumErrs);

```

```

11348     UPDATE_MIB(o1, tcpInErrs, o2->tcpInErrs);
11349     UPDATE_MIB(o1, udpNoPorts, o2->udpNoPorts);
11350 }

11352 void
11353 ip_mib2_add_icmp6_stats(mib2_ipv6IfIcmpEntry_t *o1, mib2_ipv6IfIcmpEntry_t *o2)
11354 {
11355     UPDATE_MIB(o1, ipv6IfIcmpInMsgs, o2->ipv6IfIcmpInMsgs);
11356     UPDATE_MIB(o1, ipv6IfIcmpInErrors, o2->ipv6IfIcmpInErrors);
11357     UPDATE_MIB(o1, ipv6IfIcmpInDestUnreachs, o2->ipv6IfIcmpInDestUnreachs);
11358     UPDATE_MIB(o1, ipv6IfIcmpInAdminProhibs, o2->ipv6IfIcmpInAdminProhibs);
11359     UPDATE_MIB(o1, ipv6IfIcmpInTimeExcds, o2->ipv6IfIcmpInTimeExcds);
11360     UPDATE_MIB(o1, ipv6IfIcmpInParmProblems, o2->ipv6IfIcmpInParmProblems);
11361     UPDATE_MIB(o1, ipv6IfIcmpInPktTooBigs, o2->ipv6IfIcmpInPktTooBigs);
11362     UPDATE_MIB(o1, ipv6IfIcmpInEchos, o2->ipv6IfIcmpInEchos);
11363     UPDATE_MIB(o1, ipv6IfIcmpInEchoReplies, o2->ipv6IfIcmpInEchoReplies);
11364     UPDATE_MIB(o1, ipv6IfIcmpInRouterSolicits,
11365               o2->ipv6IfIcmpInRouterSolicits);
11366     UPDATE_MIB(o1, ipv6IfIcmpInRouterAdvertisements,
11367               o2->ipv6IfIcmpInRouterAdvertisements);
11368     UPDATE_MIB(o1, ipv6IfIcmpInNeighborSolicits,
11369               o2->ipv6IfIcmpInNeighborSolicits);
11370     UPDATE_MIB(o1, ipv6IfIcmpInNeighborAdvertisements,
11371               o2->ipv6IfIcmpInNeighborAdvertisements);
11372     UPDATE_MIB(o1, ipv6IfIcmpInRedirects, o2->ipv6IfIcmpInRedirects);
11373     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembQueries,
11374               o2->ipv6IfIcmpInGroupMembQueries);
11375     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembResponses,
11376               o2->ipv6IfIcmpInGroupMembResponses);
11377     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembReductions,
11378               o2->ipv6IfIcmpInGroupMembReductions);
11379     UPDATE_MIB(o1, ipv6IfIcmpOutMsgs, o2->ipv6IfIcmpOutMsgs);
11380     UPDATE_MIB(o1, ipv6IfIcmpOutErrors, o2->ipv6IfIcmpOutErrors);
11381     UPDATE_MIB(o1, ipv6IfIcmpOutDestUnreachs,
11382               o2->ipv6IfIcmpOutDestUnreachs);
11383     UPDATE_MIB(o1, ipv6IfIcmpOutAdminProhibs,
11384               o2->ipv6IfIcmpOutAdminProhibs);
11385     UPDATE_MIB(o1, ipv6IfIcmpOutTimeExcds, o2->ipv6IfIcmpOutTimeExcds);
11386     UPDATE_MIB(o1, ipv6IfIcmpOutParmProblems,
11387               o2->ipv6IfIcmpOutParmProblems);
11388     UPDATE_MIB(o1, ipv6IfIcmpOutPktTooBigs, o2->ipv6IfIcmpOutPktTooBigs);
11389     UPDATE_MIB(o1, ipv6IfIcmpOutEchos, o2->ipv6IfIcmpOutEchos);
11390     UPDATE_MIB(o1, ipv6IfIcmpOutEchoReplies, o2->ipv6IfIcmpOutEchoReplies);
11391     UPDATE_MIB(o1, ipv6IfIcmpOutRouterSolicits,
11392               o2->ipv6IfIcmpOutRouterSolicits);
11393     UPDATE_MIB(o1, ipv6IfIcmpOutRouterAdvertisements,
11394               o2->ipv6IfIcmpOutRouterAdvertisements);
11395     UPDATE_MIB(o1, ipv6IfIcmpOutNeighborSolicits,
11396               o2->ipv6IfIcmpOutNeighborSolicits);
11397     UPDATE_MIB(o1, ipv6IfIcmpOutNeighborAdvertisements,
11398               o2->ipv6IfIcmpOutNeighborAdvertisements);
11399     UPDATE_MIB(o1, ipv6IfIcmpOutRedirects, o2->ipv6IfIcmpOutRedirects);
11400     UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembQueries,
11401               o2->ipv6IfIcmpOutGroupMembQueries);
11402     UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembResponses,
11403               o2->ipv6IfIcmpOutGroupMembResponses);
11404     UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembReductions,
11405               o2->ipv6IfIcmpOutGroupMembReductions);
11406     UPDATE_MIB(o1, ipv6IfIcmpInOverflows, o2->ipv6IfIcmpInOverflows);
11407     UPDATE_MIB(o1, ipv6IfIcmpBadHoplimit, o2->ipv6IfIcmpBadHoplimit);
11408     UPDATE_MIB(o1, ipv6IfIcmpInBadNeighborAdvertisements,
11409               o2->ipv6IfIcmpInBadNeighborAdvertisements);
11410     UPDATE_MIB(o1, ipv6IfIcmpInBadNeighborSolicitations,
11411               o2->ipv6IfIcmpInBadNeighborSolicitations);
11412     UPDATE_MIB(o1, ipv6IfIcmpInBadRedirects, o2->ipv6IfIcmpInBadRedirects);
11413     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembTotal,

```

```

11414         o2->ipv6IfIcmpInGroupMembTotal);
11415     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembBadQueries,
11416               o2->ipv6IfIcmpInGroupMembBadQueries);
11417     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembBadReports,
11418               o2->ipv6IfIcmpInGroupMembBadReports);
11419     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembOurReports,
11420               o2->ipv6IfIcmpInGroupMembOurReports);
11421 }

11422 /*
11423  * Called before the options are updated to check if this packet will
11424  * be source routed from here.
11425  * This routine assumes that the options are well formed i.e. that they
11426  * have already been checked.
11427  */
11428 */
11429 boolean_t
11430 ip_source_routed(ipha_t *ipha, ip_stack_t *ipst)
11431 {
11432     ipoptp_t     opts;
11433     uchar_t      *opt;
11434     uint8_t      optval;
11435     uint8_t      optlen;
11436     ipaddr_t     dst;

11438     if (IS_SIMPLE_IPH(ipha)) {
11439         ip2dbg(("not source routed\n"));
11440         return (B_FALSE);
11441     }
11442     dst = ipha->ipha_dst;
11443     for (optval = ipoptp_first(&opts, ipha);
11444          optval != IPOPT_EOL;
11445          optval = ipoptp_next(&opts)) {
11446         ASSERT((opts.ipoptp_flags & IPOPT_ERROR) == 0);
11447         opt = opts.ipoptp_cur;
11448         optlen = opts.ipoptp_len;
11449         ip2dbg(("ip_source_routed: opt %d, len %d\n",
11450                optval, optlen));
11451         switch (optval) {
11452             uint32_t off;
11453             case IPOPT_SSRR:
11454             case IPOPT_LSRR:
11455                 /*
11456                  * If dst is one of our addresses and there are some
11457                  * entries left in the source route return (true).
11458                  */
11459                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
11460                     ip2dbg(("ip_source_routed: not next"
11461                            " source route 0x%x\n",
11462                            ntohl(dst)));
11463                     return (B_FALSE);
11464                 }
11465                 off = opt[IPOPT_OFFSET];
11466                 off--;
11467                 if (optlen < IP_ADDR_LEN ||
11468                     off > optlen - IP_ADDR_LEN) {
11469                     /* End of source route */
11470                     ip1dbg(("ip_source_routed: end of SR\n"));
11471                     return (B_FALSE);
11472                 }
11473                 return (B_TRUE);
11474             }
11475     }
11476     ip2dbg(("not source routed\n"));
11477     return (B_FALSE);
11478 }

```

```

11480 /*
11481  * ip_unbind is called by the transports to remove a conn from
11482  * the fanout table.
11483  */
11484 void
11485 ip_unbind(conn_t *connp)
11486 {
11488     ASSERT(!MUTEX_HELD(&connp->conn_lock));
11490     if (is_system_labeled() && connp->conn_anon_port) {
11491         (void) tsol_mlp_anon(crgetzone(connp->conn_cred),
11492             connp->conn_mlp_type, connp->conn_proto,
11493             ntohs(connp->conn_lport), B_FALSE);
11494         connp->conn_anon_port = 0;
11495     }
11496     connp->conn_mlp_type = mlptSingle;
11498     ipcl_hash_remove(connp);
11499 }
11501 /*
11502  * Used for deciding the MSS size for the upper layer. Thus
11503  * we need to check the outbound policy values in the conn.
11504  */
11505 int
11506 conn_ipsec_length(conn_t *connp)
11507 {
11508     ipsec_latch_t *ipl;
11510     ipl = connp->conn_latch;
11511     if (ipl == NULL)
11512         return (0);
11514     if (connp->conn_ixa->ixa_ipsec_policy == NULL)
11515         return (0);
11517     return (connp->conn_ixa->ixa_ipsec_policy->ipsp_act->ipa_ovhd);
11518 }
11520 /*
11521  * Returns an estimate of the IPsec headers size. This is used if
11522  * we don't want to call into IPsec to get the exact size.
11523  */
11524 int
11525 ipsec_out_extra_length(ip_xmit_attr_t *ixa)
11526 {
11527     ipsec_action_t *a;
11529     if (!(ixa->ixa_flags & IXAF_IPSEC_SECURE))
11530         return (0);
11532     a = ix->ixa_ipsec_action;
11533     if (a == NULL) {
11534         ASSERT(ixa->ixa_ipsec_policy != NULL);
11535         a = ix->ixa_ipsec_policy->ipsp_act;
11536     }
11537     ASSERT(a != NULL);
11539     return (a->ipa_ovhd);
11540 }
11542 /*
11543  * If there are any source route options, return the true final
11544  * destination. Otherwise, return the destination.
11545  */

```

```

11546 ipaddr_t
11547 ip_get_dst(ipha_t *ipha)
11548 {
11549     ipoptp_t     opts;
11550     uchar_t      *opt;
11551     uint8_t      optval;
11552     uint8_t      optlen;
11553     ipaddr_t     dst;
11554     uint32_t     off;
11556     dst = ipha->ipha_dst;
11558     if (IS_SIMPLE_IPH(ipha))
11559         return (dst);
11561     for (optval = ipoptp_first(&opts, ipha);
11562         optval != IPOPT_EOL;
11563         optval = ipoptp_next(&opts)) {
11564         opt = opts.ipoptp_cur;
11565         optlen = opts.ipoptp_len;
11566         ASSERT((opts.ipoptp_flags & IPOPT_ERROR) == 0);
11567         switch (optval) {
11568             case IPOPT_SSRR:
11569             case IPOPT_LSRR:
11570                 off = opt[IPOPT_OFFSET];
11571                 /*
11572                  * If one of the conditions is true, it means
11573                  * end of options and dst already has the right
11574                  * value.
11575                  */
11576                 if (!(optlen < IP_ADDR_LEN || off > optlen - 3)) {
11577                     off = optlen - IP_ADDR_LEN;
11578                     bcopy(&opt[off], &dst, IP_ADDR_LEN);
11579                 }
11580                 return (dst);
11581             default:
11582                 break;
11583         }
11584     }
11586     return (dst);
11587 }
11589 /*
11590  * Outbound IP fragmentation routine.
11591  * Assumes the caller has checked whether or not fragmentation should
11592  * be allowed. Here we copy the DF bit from the header to all the generated
11593  * fragments.
11594  */
11595 int
11596 ip_fragment_v4(mblk_t *mp_orig, nce_t *nce, iaflags_t iaflags,
11597     uint_t pkt_len, uint32_t max_frag, uint32_t xmit_hint, zoneid_t szone,
11598     zoneid_t nolzid, pfirepostfrag_t postfragfn, uintptr_t *ixa_cookie)
11599 {
11600     int         ill;
11601     int         hdr_len;
11602     mblk_t      *hdr_mp;
11603     ipha_t      *ipha;
11604     int         ip_data_end;
11605     int         len;
11606     mblk_t      *mp = mp_orig;
11607     int         offset;
11608     ill_t       *ill = nce->nce_ill;
11609     ip_stack_t  *ipst = ill->ill_ipst;
11610     mblk_t      *carve_mp;
11611     uint32_t     frag_flag;

```

```

11612     uint_t     priority = mp->b_band;
11613     int        error = 0;

11615     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragReqds);

11617     if (pkt_len != msgdsize(mp)) {
11618         ip0dbg(("Packet length mismatch: %d, %ld\n",
11619             pkt_len, msgdsize(mp)));
11620         freemsg(mp);
11621         return (EINVAL);
11622     }

11624     if (max_frag == 0) {
11625         ipldbg(("ip_fragment_v4: max_frag is zero. Dropping packet\n"));
11626         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11627         ip_drop_output("FragFails: zero max_frag", mp, ill);
11628         freemsg(mp);
11629         return (EINVAL);
11630     }

11632     ASSERT(MBLKL(mp) >= sizeof (ipha_t));
11633     ipha = (ipha_t *)mp->b_rptr;
11634     ASSERT(ntohs(ipha->ipha_length) == pkt_len);
11635     frag_flag = ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_DF;

11637     /*
11638      * Establish the starting offset. May not be zero if we are fragging
11639      * a fragment that is being forwarded.
11640      */
11641     offset = ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_OFFSET;

11643     /* TODO why is this test needed? */
11644     if (((max_frag - ntohs(ipha->ipha_length)) & ~7) < 8) {
11645         /* TODO: notify ulp somehow */
11646         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11647         ip_drop_output("FragFails: bad starting offset", mp, ill);
11648         freemsg(mp);
11649         return (EINVAL);
11650     }

11652     hdr_len = IPH_HDR_LENGTH(ipha);
11653     ipha->ipha_hdr_checksum = 0;

11655     /*
11656      * Establish the number of bytes maximum per frag, after putting
11657      * in the header.
11658      */
11659     len = (max_frag - hdr_len) & ~7;

11661     /* Get a copy of the header for the trailing frags */
11662     hdr_mp = ip_fragment_copyhdr((uchar_t *)ipha, hdr_len, offset, ipst,
11663         mp);
11664     if (hdr_mp == NULL) {
11665         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11666         ip_drop_output("FragFails: no hdr_mp", mp, ill);
11667         freemsg(mp);
11668         return (ENOBUFS);
11669     }

11671     /* Store the starting offset, with the MoreFrag flag. */
11672     il = offset | IPH_MF | frag_flag;
11673     ipha->ipha_fragment_offset_and_flags = htons((uint16_t)il);

11675     /* Establish the ending byte offset, based on the starting offset. */
11676     offset <<= 3;
11677     ip_data_end = offset + ntohs(ipha->ipha_length) - hdr_len;

```

```

11679     /* Store the length of the first fragment in the IP header. */
11680     il = len + hdr_len;
11681     ASSERT(il <= IP_MAXPACKET);
11682     ipha->ipha_length = htons((uint16_t)il);

11684     /*
11685      * Compute the IP header checksum for the first frag. We have to
11686      * watch out that we stop at the end of the header.
11687      */
11688     ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

11690     /*
11691      * Now carve off the first frag. Note that this will include the
11692      * original IP header.
11693      */
11694     if (! (mp = ip_carve_mp(&mp_orig, il))) {
11695         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11696         ip_drop_output("FragFails: could not carve mp", mp_orig, ill);
11697         freeb(hdr_mp);
11698         freemsg(mp_orig);
11699         return (ENOBUFS);
11700     }

11702     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragCreates);

11704     error = postfragfn(mp, nce, ixaflags, il, xmit_hint, szone, nolzid,
11705         ixa_cookie);
11706     if (error != 0 && error != EWOLDBLOCK) {
11707         /* No point in sending the other fragments */
11708         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11709         ip_drop_output("FragFails: postfragfn failed", mp_orig, ill);
11710         freeb(hdr_mp);
11711         freemsg(mp_orig);
11712         return (error);
11713     }

11715     /* No need to redo state machine in loop */
11716     ixaflags &= ~IXAF_REACH_CONF;

11718     /* Advance the offset to the second frag starting point. */
11719     offset += len;
11720     /*
11721      * Update hdr_len from the copied header - there might be less options
11722      * in the later fragments.
11723      */
11724     hdr_len = IPH_HDR_LENGTH(hdr_mp->b_rptr);
11725     /* Loop until done. */
11726     for (;;) {
11727         uint16_t     offset_and_flags;
11728         uint16_t     ip_len;

11730         if (ip_data_end - offset > len) {
11731             /*
11732              * Carve off the appropriate amount from the original
11733              * datagram.
11734              */
11735             if (! (carve_mp = ip_carve_mp(&mp_orig, len))) {
11736                 mp = NULL;
11737                 break;
11738             }
11739             /*
11740              * More frags after this one. Get another copy
11741              * of the header.
11742              */
11743             if (carve_mp->b_datap->db_ref == 1 &&

```



```

11744     hdr_mp->b_wptr - hdr_mp->b_rptr <
11745     carve_mp->b_rptr - carve_mp->b_datap->db_base) {
11746     /* Inline IP header */
11747     carve_mp->b_rptr -= hdr_mp->b_wptr -
11748     hdr_mp->b_rptr;
11749     bcopy(hdr_mp->b_rptr, carve_mp->b_rptr,
11750     hdr_mp->b_wptr - hdr_mp->b_rptr);
11751     mp = carve_mp;
11752 } else {
11753     if (!(mp = copyb(hdr_mp))) {
11754         freemsg(carve_mp);
11755         break;
11756     }
11757     /* Get priority marking, if any. */
11758     mp->b_band = priority;
11759     mp->b_cont = carve_mp;
11760 }
11761 ipha = (ipha_t *)mp->b_rptr;
11762 offset_and_flags = IPH_MF;
11763 } else {
11764     /*
11765     * Last frag. Consume the header. Set len to
11766     * the length of this last piece.
11767     */
11768     len = ip_data_end - offset;
11769
11770     /*
11771     * Carve off the appropriate amount from the original
11772     * datagram.
11773     */
11774     if (!(carve_mp = ip_carve_mp(&mp_orig, len))) {
11775         mp = NULL;
11776         break;
11777     }
11778     if (carve_mp->b_datap->db_ref == 1 &&
11779     hdr_mp->b_wptr - hdr_mp->b_rptr <
11780     carve_mp->b_rptr - carve_mp->b_datap->db_base) {
11781         /* Inline IP header */
11782         carve_mp->b_rptr -= hdr_mp->b_wptr -
11783         hdr_mp->b_rptr;
11784         bcopy(hdr_mp->b_rptr, carve_mp->b_rptr,
11785         hdr_mp->b_wptr - hdr_mp->b_rptr);
11786         mp = carve_mp;
11787         freeb(hdr_mp);
11788         hdr_mp = mp;
11789     } else {
11790         mp = hdr_mp;
11791         /* Get priority marking, if any. */
11792         mp->b_band = priority;
11793         mp->b_cont = carve_mp;
11794     }
11795     ipha = (ipha_t *)mp->b_rptr;
11796     /* A frag of a frag might have IPH_MF non-zero */
11797     offset_and_flags =
11798     ntohs(ipha->ipha_fragment_offset_and_flags) &
11799     IPH_MF;
11800 }
11801 offset_and_flags |= (uint16_t)(offset >> 3);
11802 offset_and_flags |= (uint16_t)frag_flag;
11803 /* Store the offset and flags in the IP header. */
11804 ipha->ipha_fragment_offset_and_flags = htons(offset_and_flags);
11805
11806 /* Store the length in the IP header. */
11807 ip_len = (uint16_t)(len + hdr_len);
11808 ipha->ipha_length = htons(ip_len);

```

```

11810     /*
11811     * Set the IP header checksum. Note that mp is just
11812     * the header, so this is easy to pass to ip_csum.
11813     */
11814     ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
11815
11816     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragCreates);
11817
11818     error = postfragfn(mp, nce, ixaflags, ip_len, xmit_hint, szone,
11819     nolzid, ixa_cookie);
11820     /* All done if we just consumed the hdr_mp. */
11821     if (mp == hdr_mp) {
11822         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragOKs);
11823         return (error);
11824     }
11825     if (error != 0 && error != EWOULDBLOCK) {
11826         DTRACE_PROBE2(ip_xmit_frag_fail, ill_t *, ill,
11827         mblk_t *, hdr_mp);
11828         /* No point in sending the other fragments */
11829         break;
11830     }
11831
11832     /* Otherwise, advance and loop. */
11833     offset += len;
11834 }
11835 /* Clean up following allocation failure. */
11836 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11837 ip_drop_output("FragFails: loop ended", NULL, ill);
11838 if (mp != hdr_mp)
11839     freeb(hdr_mp);
11840 if (mp != mp_orig)
11841     freemsg(mp_orig);
11842 return (error);
11843 }
11844
11845 /*
11846 * Copy the header plus those options which have the copy bit set
11847 */
11848 static mblk_t *
11849 ip_fragment_copyhdr(uchar_t *rptr, int hdr_len, int offset, ip_stack_t *ipst,
11850 mblk_t *src)
11851 {
11852     mblk_t *mp;
11853     uchar_t *up;
11854
11855     /*
11856     * Quick check if we need to look for options without the copy bit
11857     * set
11858     */
11859     mp = allocb_tmpl(ipst->ips_ip_wroff_extra + hdr_len, src);
11860     if (!mp)
11861         return (mp);
11862     mp->b_rptr += ipst->ips_ip_wroff_extra;
11863     if (hdr_len == IP_SIMPLE_HDR_LENGTH || offset != 0) {
11864         bcopy(rptr, mp->b_rptr, hdr_len);
11865         mp->b_wptr += hdr_len + ipst->ips_ip_wroff_extra;
11866         return (mp);
11867     }
11868     up = mp->b_rptr;
11869     bcopy(rptr, up, IP_SIMPLE_HDR_LENGTH);
11870     up += IP_SIMPLE_HDR_LENGTH;
11871     rptr += IP_SIMPLE_HDR_LENGTH;
11872     hdr_len -= IP_SIMPLE_HDR_LENGTH;
11873     while (hdr_len > 0) {
11874         uint32_t optval;
11875         uint32_t optlen;

```

```

11877         optval = *rptr;
11878         if (optval == IPOPT_EOL)
11879             break;
11880         if (optval == IPOPT_NOP)
11881             optlen = 1;
11882         else
11883             optlen = rptr[1];
11884         if (optval & IPOPT_COPY) {
11885             bcopy(rptr, up, optlen);
11886             up += optlen;
11887         }
11888         rptr += optlen;
11889         hdr_len -= optlen;
11890     }
11891     /*
11892     * Make sure that we drop an even number of words by filling
11893     * with EOL to the next word boundary.
11894     */
11895     for (hdr_len = up - (mp->b_rptr + IP_SIMPLE_HDR_LENGTH);
11896          hdr_len & 0x3; hdr_len++)
11897         *up++ = IPOPT_EOL;
11898     mp->b_wptr = up;
11899     /* Update header length */
11900     mp->b_rptr[0] = (uint8_t)((IP_VERSION << 4) | ((up - mp->b_rptr) >> 2));
11901     return (mp);
11902 }

11904 /*
11905 * Update any source route, record route, or timestamp options when
11906 * sending a packet back to ourselves.
11907 * Check that we are at end of strict source route.
11908 * The options have been sanity checked by ip_output_options().
11909 */
11910 void
11911 ip_output_local_options(ipha_t *ipha, ip_stack_t *ipst)
11912 {
11913     ipoptp_t      opts;
11914     uchar_t       *opt;
11915     uint8_t       optval;
11916     uint8_t       optlen;
11917     ipaddr_t      dst;
11918     uint32_t       ts;
11919     timestruc_t   now;

11921     for (optval = ipoptp_first(&opts, ipha);
11922          optval != IPOPT_EOL;
11923          optval = ipoptp_next(&opts)) {
11924         opt = opts.ipoptp_cur;
11925         optlen = opts.ipoptp_len;
11926         ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
11927         switch (optval) {
11928             uint32_t off;
11929         case IPOPT_SSRR:
11930         case IPOPT_LSRR:
11931             off = opt[IPOPT_OFFSET];
11932             off--;
11933             if (optlen < IP_ADDR_LEN ||
11934                 off > optlen - IP_ADDR_LEN) {
11935                 /* End of source route */
11936                 break;
11937             }
11938             /*
11939             * This will only happen if two consecutive entries
11940             * in the source route contains our address or if
11941             * it is a packet with a loose source route which

```

```

11942         * reaches us before consuming the whole source route
11943         */
11944     if (optval == IPOPT_SSRR) {
11945         return;
11946     }
11947     /*
11948     * Hack: instead of dropping the packet truncate the
11949     * source route to what has been used by filling the
11950     * rest with IPOPT_NOP.
11951     */
11952     opt[IPOPT_OLEN] = (uint8_t)off;
11953     while (off < optlen) {
11954         opt[off++] = IPOPT_NOP;
11955     }
11956     break;
11957 case IPOPT_RR:
11958     off = opt[IPOPT_OFFSET];
11959     off--;
11960     if (optlen < IP_ADDR_LEN ||
11961         off > optlen - IP_ADDR_LEN) {
11962         /* No more room - ignore */
11963         ipldbg(("ip_output_local_options: end of RR\n"));
11964         break;
11965     }
11966     dst = htonl(INADDR_LOOPBACK);
11967     bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
11968     opt[IPOPT_OFFSET] += IP_ADDR_LEN;
11969     break;
11970 case IPOPT_TS:
11971     /* Insert timestamp if there is room */
11972     switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
11973         case IPOPT_TS_TSONLY:
11974             off = IPOPT_TS_TIMELEN;
11975             break;
11976         case IPOPT_TS_PRESPEC:
11977         case IPOPT_TS_PRESPEC_RFC791:
11978             /* Verify that the address matched */
11979             off = opt[IPOPT_OFFSET] - 1;
11980             bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
11981             if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
11982                 /* Not for us */
11983                 break;
11984             }
11985             /* FALLTHRU */
11986         case IPOPT_TS_TSANDADDR:
11987             off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
11988             break;
11989     default:
11990         /*
11991         * ip_*put_options should have already
11992         * dropped this packet.
11993         */
11994         cmn_err(CE_PANIC, "ip_output_local_options: "
11995             "unknown IT - bug in ip_output_options?\n");
11996         return; /* Keep "lint" happy */
11997     }
11998     if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
11999         /* Increase overflow counter */
12000         off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
12001         opt[IPOPT_POS_OV_FLG] = (uint8_t)
12002             (opt[IPOPT_POS_OV_FLG] & 0x0F) |
12003             (off << 4);
12004         break;
12005     }

```

```

12008         off = opt[IPOPT_OFFSET] - 1;
12009         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
12010         case IPOPT_TS_PRESPEC:
12011         case IPOPT_TS_PRESPEC_RFC791:
12012         case IPOPT_TS_TSANDADDR:
12013             dst = htonl(INADDR_LOOPBACK);
12014             bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
12015             opt[IPOPT_OFFSET] += IP_ADDR_LEN;
12016             /* FALLTHRU */
12017         case IPOPT_TS_TSONLY:
12018             off = opt[IPOPT_OFFSET] - 1;
12019             /* Compute # of milliseconds since midnight */
12020             gethrestime(&now);
12021             ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
12022                 now.tv_nsec / (NANOSEC / MILLISEC);
12023             bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
12024             opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
12025             break;
12026         }
12027     }
12028 }
12029 }
12030 }

12032 /*
12033  * Prepend an M_DATA fastpath header, and if none present prepend a
12034  * DL_UNITDATA_REQ. Frees the mblk on failure.
12035  *
12036  * nce_dlur_mp and nce_fp_mp can not disappear once they have been set.
12037  * If there is a change to them, the nce will be deleted (condemned) and
12038  * a new nce_t will be created when packets are sent. Thus we need no locks
12039  * to access those fields.
12040  *
12041  * We preserve b_band to support IPQoS. If a DL_UNITDATA_REQ is prepended
12042  * we place b_band in dl_priority.dl_max.
12043  */
12044 static mblk_t *
12045 ip_xmit_attach_llhdr(mblk_t *mp, nce_t *nce)
12046 {
12047     uint_t hlen;
12048     mblk_t *mpl;
12049     uint_t priority;
12050     uchar_t *rptr;

12052     rptr = mp->b_rptr;

12054     ASSERT(DB_TYPE(mp) == M_DATA);
12055     priority = mp->b_band;

12057     ASSERT(nce != NULL);
12058     if ((mpl = nce->nce_fp_mp) != NULL) {
12059         hlen = MBLKL(mpl);
12060         /*
12061          * Check if we have enough room to prepend fastpath
12062          * header
12063          */
12064         if (hlen != 0 && (rptr - mp->b_datap->db_base) >= hlen) {
12065             rptr -= hlen;
12066             bcopy(mpl->b_rptr, rptr, hlen);
12067             /*
12068              * Set the b_rptr to the start of the link layer
12069              * header
12070              */
12071             mp->b_rptr = rptr;
12072             return (mp);
12073         }

```

```

12074         mpl = copyb(mpl);
12075         if (mpl == NULL) {
12076             ill_t *ill = nce->nce_ill;

12078             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12079             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12080             freemsg(mp);
12081             return (NULL);
12082         }
12083         mpl->b_band = priority;
12084         mpl->b_cont = mp;
12085         DB_CKSUMSTART(mpl) = DB_CKSUMSTART(mp);
12086         DB_CKSUMSTUFF(mpl) = DB_CKSUMSTUFF(mp);
12087         DB_CKSUMEND(mpl) = DB_CKSUMEND(mp);
12088         DB_CKSUMFLAGS(mpl) = DB_CKSUMFLAGS(mp);
12089         DB_LSOMSS(mpl) = DB_LSOMSS(mp);
12090         DTRACE_PROBE1(ip_xmit_copyb, (mblk_t *), mpl);
12091         /*
12092          * XXX disable ICK_VALID and compute checksum
12093          * here; can happen if nce_fp_mp changes and
12094          * it can't be copied now due to insufficient
12095          * space. (unlikely, fp mp can change, but it
12096          * does not increase in length)
12097          */
12098         return (mpl);
12099     }
12100     mpl = copyb(nce->nce_dlur_mp);

12102     if (mpl == NULL) {
12103         ill_t *ill = nce->nce_ill;

12105         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12106         ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12107         freemsg(mp);
12108         return (NULL);
12109     }
12110     mpl->b_cont = mp;
12111     if (priority != 0) {
12112         mpl->b_band = priority;
12113         ((dl_unitdata_req_t *) (mpl->b_rptr))->dl_priority.dl_max =
12114             priority;
12115     }
12116     return (mpl);
12117 #undef rptr
12118 }

12120 /*
12121  * Finish the outbound IPsec processing. This function is called from
12122  * ipsec_out_process() if the IPsec packet was processed
12123  * synchronously, or from {ah,esp}_kcf_callback_outbound() if it was processed
12124  * asynchronously.
12125  *
12126  * This is common to IPv4 and IPv6.
12127  */
12128 int
12129 ip_output_post_ipsec(mblk_t *mp, ip_xmit_attr_t *ixa)
12130 {
12131     iaflags_t         ixaflags = ix->ixa_flags;
12132     uint_t             pktlen;

12135     /* AH/ESP don't update ixaflags when they modify the packet */
12136     if (ixaflags & IXAF_IS_IPV4) {
12137         ipha_t         *ipha = (ipha_t *) mp->b_rptr;

12139         ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);

```

```

12140         pktlen = ntohs(ip6h->ip6h_length);
12141     } else {
12142         ip6_t             *ip6h = (ip6_t *)mp->b_rptr;

12144         ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);
12145         pktlen = ntohs(ip6h->ip6h_plen) + IPV6_HDR_LEN;
12146     }

12148     /*
12149     * We release any hard reference on the SAs here to make
12150     * sure the SAs can be garbage collected. ipsr_sa has a soft reference
12151     * on the SAs.
12152     * If in the future we want the hard latching of the SAs in the
12153     * ip_xmit_attr_t then we should remove this.
12154     */
12155     if (ixa->ixa_ipsec_esp_sa != NULL) {
12156         IPSA_REFRELE(ip6h->ip6h_esp_sa);
12157         ip6h->ip6h_esp_sa = NULL;
12158     }
12159     if (ixa->ixa_ipsec_ah_sa != NULL) {
12160         IPSA_REFRELE(ip6h->ip6h_ah_sa);
12161         ip6h->ip6h_ah_sa = NULL;
12162     }

12164     /* Do we need to fragment? */
12165     if ((ixa->ixa_flags & IXAF_IPV6_ADD_FRAGHDR) ||
12166         pktlen > ip6h->ip6h_fragsize) {
12167         if (ixaflags & IXAF_IS_IPV4) {
12168             ASSERT(!(ixa->ixa_flags & IXAF_IPV6_ADD_FRAGHDR));
12169             /*
12170             * We check for the DF case in ipsec_out_process
12171             * hence this only handles the non-DF case.
12172             */
12173             return (ip_fragment_v4(mp, ip6h->ip6h_nce, ip6h->ip6h_flags,
12174                 pktlen, ip6h->ip6h_fragsize,
12175                 ip6h->ip6h_xmit_hint, ip6h->ip6h_zoneid,
12176                 ip6h->ip6h_no_loop_zoneid, ip6h->ip6h_postfragfn,
12177                 ip6h->ip6h_cookie));
12178         } else {
12179             mp = ip_fraghdr_add_v6(mp, ip6h->ip6h_ident, ip6h);
12180             if (mp == NULL) {
12181                 /* MIB and ip_drop_output already done */
12182                 return (ENOMEM);
12183             }
12184             pktlen += sizeof (ip6_frag_t);
12185             if (pktlen > ip6h->ip6h_fragsize) {
12186                 return (ip_fragment_v6(mp, ip6h->ip6h_nce,
12187                     ip6h->ip6h_flags, pktlen,
12188                     ip6h->ip6h_fragsize, ip6h->ip6h_xmit_hint,
12189                     ip6h->ip6h_zoneid, ip6h->ip6h_no_loop_zoneid,
12190                     ip6h->ip6h_postfragfn, ip6h->ip6h_cookie));
12191             }
12192         }
12193     }
12194     return ((ixa->ixa_postfragfn)(mp, ip6h->ip6h_nce, ip6h->ip6h_flags,
12195         pktlen, ip6h->ip6h_xmit_hint, ip6h->ip6h_zoneid,
12196         ip6h->ip6h_no_loop_zoneid, NULL));
12197 }

12199 /*
12200 * Finish the inbound IPsec processing. This function is called from
12201 * ipsec_out_process() if the IPsec packet was processed
12202 * synchronously, or from {ah,esp}_kcf_callback_outbound() if it was processed
12203 * asynchronously.
12204 *
12205 * This is common to IPv4 and IPv6.

```

```

12206     /*
12207     void
12208     ip_input_post_ipsec(mblk_t *mp, ip_recv_attr_t *ira)
12209     {
12210         iaflags_t         iraflags = ira->ira_flags;

12212         /* Length might have changed */
12213         if (iraflags & IRAF_IS_IPV4) {
12214             ipha_t         *ipha = (ipha_t *)mp->b_rptr;

12216             ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);
12217             ira->ira_pktlen = ntohs(ipha->ipha_length);
12218             ira->ira_ip_hdr_length = IPH_HDR_LENGTH(ipha);
12219             ira->ira_protocol = ipha->ipha_protocol;

12221             ip_fanout_v4(mp, ipha, ira);
12222         } else {
12223             ip6_t             *ip6h = (ip6_t *)mp->b_rptr;
12224             uint8_t          *nexthdrp;

12226             ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);
12227             ira->ira_pktlen = ntohs(ip6h->ip6h_plen) + IPV6_HDR_LEN;
12228             if (!ip_hdr_length_nexthdr_v6(mp, ip6h, &ira->ira_ip_hdr_length,
12229                 &nexthdrp) {
12230                 /* Malformed packet */
12231                 BUMP_MIB(ira->ira_ill->ill_ip_mib, ipIfStatsInDiscards);
12232                 ip_drop_input("ipIfStatsInDiscards", mp, ira->ira_ill);
12233                 freemsg(mp);
12234                 return;
12235             }
12236             ira->ira_protocol = *nexthdrp;
12237             ip_fanout_v6(mp, ip6h, ira);
12238         }
12239     }

12241     /*
12242     * Select which AH & ESP SA's to use (if any) for the outbound packet.
12243     *
12244     * If this function returns B_TRUE, the requested SA's have been filled
12245     * into the ipsec_*_sa pointers.
12246     *
12247     * If the function returns B_FALSE, the packet has been "consumed", most
12248     * likely by an ACQUIRE sent up via PF_KEY to a key management daemon.
12249     *
12250     * The SA references created by the protocol-specific "select"
12251     * function will be released in ip_output_post_ipsec.
12252     */
12253     static boolean_t
12254     ipsec_out_select_sa(mblk_t *mp, ip_xmit_attr_t *ixa)
12255     {
12256         boolean_t need_ah_acquire = B_FALSE, need_esp_acquire = B_FALSE;
12257         ipsec_policy_t *pp;
12258         ipsec_action_t *ap;

12260         ASSERT(ip6h->ip6h_flags & IXAF_IPSEC_SECURE);
12261         ASSERT((ixa->ixa_ipsec_policy != NULL) ||
12262             (ixa->ixa_ipsec_action != NULL));

12264         ap = ip6h->ip6h_ipsec_action;
12265         if (ap == NULL) {
12266             pp = ip6h->ip6h_ipsec_policy;
12267             ASSERT(pp != NULL);
12268             ap = pp->ipsec_act;
12269             ASSERT(ap != NULL);
12270         }

```

```

12272  /*
12273  * We have an action.  now, let's select SA's.
12274  * A side effect of setting ixa_ipsec_*_sa is that it will
12275  * be cached in the conn_t.
12276  */
12277  if (ap->ipa_want_esp) {
12278      if (ixa->ixa_ipsec_esp_sa == NULL) {
12279          need_esp_acquire = !ipsec_outbound_sa(mp, ixa,
12280              IPPROTO_ESP);
12281      }
12282      ASSERT(need_esp_acquire || ixa->ixa_ipsec_esp_sa != NULL);
12283  }
12284
12285  if (ap->ipa_want_ah) {
12286      if (ixa->ixa_ipsec_ah_sa == NULL) {
12287          need_ah_acquire = !ipsec_outbound_sa(mp, ixa,
12288              IPPROTO_AH);
12289      }
12290      ASSERT(need_ah_acquire || ixa->ixa_ipsec_ah_sa != NULL);
12291      /*
12292       * The ESP and AH processing order needs to be preserved
12293       * when both protocols are required (ESP should be applied
12294       * before AH for an outbound packet).  Force an ESP ACQUIRE
12295       * when both ESP and AH are required, and an AH ACQUIRE
12296       * is needed.
12297       */
12298      if (ap->ipa_want_esp && need_ah_acquire)
12299          need_esp_acquire = B_TRUE;
12300  }
12301
12302  /*
12303  * Send an ACQUIRE (extended, regular, or both) if we need one.
12304  * Release SAs that got referenced, but will not be used until we
12305  * acquire_all_of the SAs we need.
12306  */
12307  if (need_ah_acquire || need_esp_acquire) {
12308      if (ixa->ixa_ipsec_ah_sa != NULL) {
12309          IPSA_REFRELE(ixa->ixa_ipsec_ah_sa);
12310          ixa->ixa_ipsec_ah_sa = NULL;
12311      }
12312      if (ixa->ixa_ipsec_esp_sa != NULL) {
12313          IPSA_REFRELE(ixa->ixa_ipsec_esp_sa);
12314          ixa->ixa_ipsec_esp_sa = NULL;
12315      }
12316
12317      sadb_acquire(mp, ixa, need_ah_acquire, need_esp_acquire);
12318      return (B_FALSE);
12319  }
12320
12321  return (B_TRUE);
12322 }
12323
12324 /*
12325  * Handle IPsec output processing.
12326  * This function is only entered once for a given packet.
12327  * We try to do things synchronously, but if we need to have user-level
12328  * set up SAs, or ESP or AH uses asynchronous KEF, then the operation
12329  * will be completed
12330  * - when the SAs are added in esp_add_sa_finish/ah_add_sa_finish
12331  * - when asynchronous ESP is done it will do AH
12332  *
12333  * In all cases we come back in ip_output_post_ipsec() to fragment and
12334  * send out the packet.
12335  */
12336 int
12337 ipsec_out_process(mblk_t *mp, ip_xmit_attr_t *ixa)

```

```

12338 {
12339     ill_t          *ill = ixa->ixa_nce->nce_ill;
12340     ip_stack_t     *ipst = ixa->ixa_ipst;
12341     ipsec_stack_t  *ipss;
12342     ipsec_policy_t *pp;
12343     ipsec_action_t *ap;
12344
12345     ASSERT(ixa->ixa_flags & IXAF_IPSEC_SECURE);
12346
12347     ASSERT((ixa->ixa_ipsec_policy != NULL) ||
12348         (ixa->ixa_ipsec_action != NULL));
12349
12350     ipss = ipst->ips_netstack->netstack_ipsec;
12351     if (!ipsec_loaded(ipss)) {
12352         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12353         ip_drop_packet(mp, B_TRUE, ill,
12354             DROPPER(ipss, ipds_ip_ipsec_not_loaded),
12355             &ipss->ipsec_droppper);
12356         return (ENOTSUP);
12357     }
12358
12359     ap = ixa->ixa_ipsec_action;
12360     if (ap == NULL) {
12361         pp = ixa->ixa_ipsec_policy;
12362         ASSERT(pp != NULL);
12363         ap = pp->ipsp_act;
12364         ASSERT(ap != NULL);
12365     }
12366
12367     /* Handle explicit drop action and bypass. */
12368     switch (ap->ipa_act.ipa_type) {
12369     case IPSEC_ACT_DISCARD:
12370     case IPSEC_ACT_REJECT:
12371         ip_drop_packet(mp, B_FALSE, ill,
12372             DROPPER(ipss, ipds_spd_explicit), &ipss->ipsec_spd_droppper);
12373         return (EHOSTUNREACH); /* IPsec policy failure */
12374     case IPSEC_ACT_BYPASS:
12375         return (ip_output_post_ipsec(mp, ixa));
12376     }
12377
12378     /*
12379      * The order of processing is first insert a IP header if needed.
12380      * Then insert the ESP header and then the AH header.
12381      */
12382     if ((ixa->ixa_flags & IXAF_IS_IPV4) && ap->ipa_want_se) {
12383         /*
12384          * First get the outer IP header before sending
12385          * it to ESP.
12386          */
12387         ipha_t *oipha, *iipha;
12388         mblk_t *outer_mp, *inner_mp;
12389
12390         if ((outer_mp = allocb(sizeof (ipha_t), BPRI_HI)) == NULL) {
12391             (void) mi_strlog(ill->ill_rq, 0,
12392                 SL_ERROR|SL_TRACE|SL_CONSOLE,
12393                 "ipsec_out_process: "
12394                 "Self-Encapsulation failed: Out of memory\n");
12395             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12396             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12397             freemsg(mp);
12398             return (ENOBUFS);
12399         }
12400         inner_mp = mp;
12401         ASSERT(inner_mp->b_datap->db_type == M_DATA);
12402         oipha = (ipha_t *)outer_mp->b_rptr;
12403         iipha = (ipha_t *)inner_mp->b_rptr;

```

```

12404     *oipha = *iipha;
12405     outer_mp->b_wptr += sizeof (ipha_t);
12406     oipha->ipha_length = htons(ntohs(iipha->ipha_length) +
12407         sizeof (ipha_t));
12408     oipha->ipha_protocol = IPPROTO_ENCAP;
12409     oipha->ipha_version_and_hdr_length =
12410         IP_SIMPLE_HDR_VERSION;
12411     oipha->ipha_hdr_checksum = 0;
12412     oipha->ipha_hdr_checksum = ip_csum_hdr(oipha);
12413     outer_mp->b_cont = inner_mp;
12414     mp = outer_mp;

12416     ixa->ixa_flags |= IXAF_IPSEC_TUNNEL;
12417 }

12419 /* If we need to wait for a SA then we can't return any errno */
12420 if (((ap->ipa_want_ah && (ixa->ixa_ipsec_ah_sa == NULL)) ||
12421     (ap->ipa_want_esp && (ixa->ixa_ipsec_esp_sa == NULL))) &&
12422     !ipsec_out_select_sa(mp, ixa))
12423     return (0);

12425 /*
12426  * By now, we know what SA's to use. Toss over to ESP & AH
12427  * to do the heavy lifting.
12428  */
12429 if (ap->ipa_want_esp) {
12430     ASSERT(ixa->ixa_ipsec_esp_sa != NULL);

12432     mp = ixa->ixa_ipsec_esp_sa->ipsa_output_func(mp, ixa);
12433     if (mp == NULL) {
12434         /*
12435          * Either it failed or is pending. In the former case
12436          * ipIfStatsInDiscards was increased.
12437          */
12438         return (0);
12439     }
12440 }

12442 if (ap->ipa_want_ah) {
12443     ASSERT(ixa->ixa_ipsec_ah_sa != NULL);

12445     mp = ixa->ixa_ipsec_ah_sa->ipsa_output_func(mp, ixa);
12446     if (mp == NULL) {
12447         /*
12448          * Either it failed or is pending. In the former case
12449          * ipIfStatsInDiscards was increased.
12450          */
12451         return (0);
12452     }
12453 }
12454 /*
12455  * We are done with IPsec processing. Send it over
12456  * the wire.
12457  */
12458 return (ip_output_post_ipsec(mp, ixa));
12459 }

12461 /*
12462  * ioctls that go through a down/up sequence may need to wait for the down
12463  * to complete. This involves waiting for the ire and ipif refents to go down
12464  * to zero. Subsequently the ioctl is restarted from ipif_ill_refrele_tail.
12465  */
12466 /* ARGSUSED */
12467 void
12468 ip_reprocess_ioctl(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
12469 {

```

```

12470     struct iocblk *iocp;
12471     mblk_t *mpl;
12472     ip_ioctl_cmd_t *ipip;
12473     int err;
12474     sin_t *sin;
12475     struct lifreq *lifr;
12476     struct ifreq *ifr;

12478     iocp = (struct iocblk *)mp->b_rptr;
12479     ASSERT(ipsq != NULL);
12480     /* Existence of mpl verified in ip_wput_nondata */
12481     mpl = mp->b_cont->b_cont;
12482     ipip = ip_ioctl_lookup(iocp->ioc_cmd);
12483     if (ipip->ipi_cmd == SIOCSLIFNAME || ipip->ipi_cmd == IF_UNITSEL) {
12484         /*
12485          * Special case where ipx_current_ipif is not set:
12486          * ill_phyint_reinit merged the v4 and v6 into a single ipsq.
12487          * We are here as were not able to complete the operation in
12488          * ipif_set_values because we could not become exclusive on
12489          * the new ipsq.
12490          */
12491         ill_t *ill = q->q_ptr;
12492         ipsq_current_start(ipsq, ill->ill_ipif, ipip->ipi_cmd);
12493     }
12494     ASSERT(ipsq->ipsq_xop->ipx_current_ipif != NULL);

12496     if (ipip->ipi_cmd_type == IF_CMD) {
12497         /* This a old style SIOC[GS]IF* command */
12498         ifr = (struct ifreq *)mpl->b_rptr;
12499         sin = (sin_t *)&ifr->ifr_addr;
12500     } else if (ipip->ipi_cmd_type == LIF_CMD) {
12501         /* This a new style SIOC[GS]LIF* command */
12502         lifr = (struct lifreq *)mpl->b_rptr;
12503         sin = (sin_t *)&lifr->lifr_addr;
12504     } else {
12505         sin = NULL;
12506     }

12508     err = (*ipip->ipi_func_restart)(ipsq->ipsq_xop->ipx_current_ipif, sin,
12509         q, mp, ipip, mpl->b_rptr);

12511     DTRACE_PROBE4(ipif_ioctl, char *, "ip_reprocess_ioctl finish",
12512         int, ipip->ipi_cmd,
12513         ill_t *, ipsq->ipsq_xop->ipx_current_ipif->ipif_ill,
12514         ipif_t *, ipsq->ipsq_xop->ipx_current_ipif);

12516     ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), ipsq);
12517 }

12519 /*
12520  * ioctl processing
12521  *
12522  * ioctl processing starts with ip_ioctl_copyin_setup(), which looks up
12523  * the ioctl command in the ioctl tables, determines the copyin data size
12524  * from the ipi_copyin_size field, and does an mi_copyin() of that size.
12525  *
12526  * ioctl processing then continues when the M_IOCTLDATA makes its way down to
12527  * ip_wput_nondata(). The ioctl is looked up again in the ioctl table, its
12528  * associated 'conn' is refheld till the end of the ioctl and the general
12529  * ioctl processing function ip_process_ioctl() is called to extract the
12530  * arguments and process the ioctl. To simplify extraction, ioctl commands
12531  * are "typed" based on the arguments they take (e.g., LIF_CMD which takes a
12532  * 'struct lifreq'), and a common extract function (e.g., ip_extract_lifreq())
12533  * is used to extract the ioctl's arguments.
12534  *
12535  * ip_process_ioctl determines if the ioctl needs to be serialized, and if

```

```

12536 * so goes thru the serialization primitive ipsq_try_enter. Then the
12537 * appropriate function to handle the ioctl is called based on the entry in
12538 * the ioctl table. ioctl completion is encapsulated in ip_ioctl_finish
12539 * which also refreleases the 'conn' that was refheld at the start of the
12540 * ioctl. Finally ipsq_exit is called if needed to exit the ipsq.
12541 *
12542 * Many exclusive ioctls go thru an internal down up sequence as part of
12543 * the operation. For example an attempt to change the IP address of an
12544 * ipif entails ipif_down, set address, ipif_up. Bringing down the interface
12545 * does all the cleanup such as deleting all ires that use this address.
12546 * Then we need to wait till all references to the interface go away.
12547 */
12548 void
12549 ip_process_ioctl(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *arg)
12550 {
12551     struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
12552     ip_ioctl_cmd_t *ipip = arg;
12553     ip_extract_func_t *extract_funcp;
12554     cmd_info_t ci;
12555     int err;
12556     boolean_t entered_ipsq = B_FALSE;

12558     ip3dbg(("ip_process_ioctl: ioctl %X\n", iocp->ioc_cmd));

12560     if (ipip == NULL)
12561         ipip = ip_sioctl_lookup(iocp->ioc_cmd);

12563     /*
12564      * SIOCLIFADDIF needs to go thru a special path since the
12565      * ill may not exist yet. This happens in the case of lo0
12566      * which is created using this ioctl.
12567      */
12568     if (ipip->ipi_cmd == SIOCLIFADDIF) {
12569         err = ip_sioctl_addif(NULL, NULL, q, mp, NULL, NULL);
12570         DTRACE_PROBE4(ipif_ioctl, char *, "ip_process_ioctl finish",
12571             int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12572         ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12573         return;
12574     }

12576     ci.ci_ipif = NULL;
12577     switch (ipip->ipi_cmd_type) {
12578     case MISC_CMD:
12579     case MSFILT_CMD:
12580         /*
12581          * All MISC_CMD ioctls come in here -- e.g. SIOCGIFCONF.
12582          */
12583         if (ipip->ipi_cmd == IF_UNITSEL) {
12584             /* ioctl comes down the ill */
12585             ci.ci_ipif = ((ill_t *)q->q_ptr)->ill_ipif;
12586             ipif_refhold(ci.ci_ipif);
12587         }
12588         err = 0;
12589         ci.ci_sin = NULL;
12590         ci.ci_sin6 = NULL;
12591         ci.ci_lifr = NULL;
12592         extract_funcp = NULL;
12593         break;

12595     case IF_CMD:
12596     case LIF_CMD:
12597         extract_funcp = ip_extract_lifreq;
12598         break;

12600     case ARP_CMD:
12601     case XARP_CMD:

```

```

12602         extract_funcp = ip_extract_arpreq;
12603         break;

12605     default:
12606         ASSERT(0);
12607     }

12609     if (extract_funcp != NULL) {
12610         err = (*extract_funcp)(q, mp, ipip, &ci);
12611         if (err != 0) {
12612             DTRACE_PROBE4(ipif_ioctl,
12613                 char *, "ip_process_ioctl finish err",
12614                 int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12615             ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12616             return;
12617         }

12619         /*
12620          * All of the extraction functions return a refheld ipif.
12621          */
12622         ASSERT(ci.ci_ipif != NULL);
12623     }

12625     if (!(ipip->ipi_flags & IPI_WR)) {
12626         /*
12627          * A return value of EINPROGRESS means the ioctl is
12628          * either queued and waiting for some reason or has
12629          * already completed.
12630          */
12631         err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip,
12632             ci.ci_lifr);
12633         if (ci.ci_ipif != NULL) {
12634             DTRACE_PROBE4(ipif_ioctl,
12635                 char *, "ip_process_ioctl finish RD",
12636                 int, ipip->ipi_cmd, ill_t *, ci.ci_ipif->ipif_ill,
12637                 ipif_t *, ci.ci_ipif);
12638             ipif_refrele(ci.ci_ipif);
12639         } else {
12640             DTRACE_PROBE4(ipif_ioctl,
12641                 char *, "ip_process_ioctl finish RD",
12642                 int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12643         }
12644         ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12645         return;
12646     }

12648     ASSERT(ci.ci_ipif != NULL);

12650     /*
12651      * If ipsq is non-NULL, we are already being called exclusively
12652      */
12653     ASSERT(ipsq == NULL || IAM_WRITER_IPSQ(ipsq));
12654     if (ipsq == NULL) {
12655         ipsq = ipsq_try_enter(ci.ci_ipif, NULL, q, mp, ip_process_ioctl,
12656             NEW_OP, B_TRUE);
12657         if (ipsq == NULL) {
12658             ipif_refrele(ci.ci_ipif);
12659             return;
12660         }
12661         entered_ipsq = B_TRUE;
12662     }
12663     /*
12664      * Release the ipif so that ipif_down and friends that wait for
12665      * references to go away are not misled about the current ipif_refcnt
12666      * values. We are writer so we can access the ipif even after releasing
12667      * the ipif.

```

```

12668      */
12669      ipif_refrele(ci.ci_ipif);

12671      ipsq_current_start(ipsq, ci.ci_ipif, ipip->ipi_cmd);

12673      /*
12674      * A return value of EINPROGRESS means the ioctl is
12675      * either queued and waiting for some reason or has
12676      * already completed.
12677      */
12678      err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip, ci.ci_lifr);

12680      DTRACE_PROBE4(ipif_ioctl, char *, "ip_process_ioctl finish WR",
12681                  int, ipip->ipi_cmd,
12682                  ill_t *, ci.ci_ipif == NULL ? NULL : ci.ci_ipif->ipif_ill,
12683                  ipif_t *, ci.ci_ipif);
12684      ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), ipsq);

12686      if (entered_ipsq)
12687          ipsq_exit(ipsq);
12688  }

12690  /*
12691  * Complete the ioctl. Typically ioctls use the mi package and need to
12692  * do mi_copyout/mi_copy_done.
12693  */
12694  void
12695  ip_ioctl_finish(queue_t *q, mblk_t *mp, int err, int mode, ipsq_t *ipsq)
12696  {
12697      conn_t *connp = NULL;

12699      if (err == EINPROGRESS)
12700          return;

12702      if (CONN_Q(q)) {
12703          connp = Q_TO_CONN(q);
12704          ASSERT(connp->conn_ref >= 2);
12705      }

12707      switch (mode) {
12708      case COPYOUT:
12709          if (err == 0)
12710              mi_copyout(q, mp);
12711          else
12712              mi_copy_done(q, mp, err);
12713          break;

12715      case NO_COPYOUT:
12716          mi_copy_done(q, mp, err);
12717          break;

12719      default:
12720          ASSERT(mode == CONN_CLOSE);      /* aborted through CONN_CLOSE */
12721          break;
12722      }

12724      /*
12725      * The conn rehold and ioctlref placed on the conn at the start of the
12726      * ioctl are released here.
12727      */
12728      if (connp != NULL) {
12729          CONN_DEC_IOCTLREF(connp);
12730          CONN_OPER_PENDING_DONE(connp);
12731      }

12733      if (ipsq != NULL)

```

```

12734          ipsq_current_finish(ipsq);
12735      }

12737      /* Handles all non data messages */
12738      void
12739      ip_wput_nondata(queue_t *q, mblk_t *mp)
12740      {
12741          mblk_t *mpl;
12742          struct iocblk *iocp;
12743          ip_ioctl_cmd_t *pip;
12744          conn_t *connp;
12745          cred_t *cr;
12746          char *proto_str;

12748          if (CONN_Q(q))
12749              connp = Q_TO_CONN(q);
12750          else
12751              connp = NULL;

12753          switch (DB_TYPE(mp)) {
12754          case M_IOCTL:
12755              /*
12756              * IOCTL processing begins in ip_siocctl_copyin_setup which
12757              * will arrange to copy in associated control structures.
12758              */
12759              ip_siocctl_copyin_setup(q, mp);
12760              return;
12761          case M_IOCTLDATA:
12762              /*
12763              * Ensure that this is associated with one of our trans-
12764              * parent ioctls. If it's not ours, discard it if we're
12765              * running as a driver, or pass it on if we're a module.
12766              */
12767              iocp = (struct iocblk *)mp->b_rptr;
12768              ipip = ip_siocctl_lookup(iocp->ioc_cmd);
12769              if (ipip == NULL) {
12770                  if (q->q_next == NULL) {
12771                      goto nak;
12772                  } else {
12773                      putnext(q, mp);
12774                  }
12775                  return;
12776              }
12777              if ((q->q_next != NULL) && !(ipip->ipi_flags & IPI_MODOK)) {
12778                  /*
12779                  * The ioctl is one we recognise, but is not consumed
12780                  * by IP as a module and we are a module, so we drop
12781                  */
12782                  goto nak;
12783              }

12785              /* IOCTL continuation following copyin or copyout. */
12786              if (mi_copy_state(q, mp, NULL) == -1) {
12787                  /*
12788                  * The copy operation failed. mi_copy_state already
12789                  * cleaned up, so we're out of here.
12790                  */
12791                  return;
12792              }
12793              /*
12794              * If we just completed a copy in, we become writer and
12795              * continue processing in ip_siocctl_copyin_done. If it
12796              * was a copy out, we call mi_copyout again. If there is
12797              * nothing more to copy out, it will complete the IOCTL.
12798              */
12799              if (MI_COPY_DIRECTION(mp) == MI_COPY_IN) {

```



```

12800         if (!(mpl = mp->b_cont) || !(mpl = mp1->b_cont)) {
12801             mi_copy_done(q, mp, EPROTO);
12802             return;
12803         }
12804         /*
12805          * Check for cases that need more copying. A return
12806          * value of 0 means a second copyin has been started,
12807          * so we return; a return value of 1 means no more
12808          * copying is needed, so we continue.
12809          */
12810         if (ipip->ipi_cmd_type == MSFILT_CMD &&
12811             MI_COPY_COUNT(mp) == 1) {
12812             if (ip_copyin_msfilter(q, mp) == 0)
12813                 return;
12814         }
12815         /*
12816          * Refhold the conn, till the ioctl completes. This is
12817          * needed in case the ioctl ends up in the pending mp
12818          * list. Every mp in the ipx_pending_mp list must have
12819          * a rehold on the conn to resume processing. The
12820          * rehold is released when the ioctl completes
12821          * (whether normally or abnormally). An ioctlref is also
12822          * placed on the conn to prevent TCP from removing the
12823          * queue needed to send the ioctl reply back.
12824          * In all cases ip_ioctl_finish is called to finish
12825          * the ioctl and release the reholds.
12826          */
12827         if (connp != NULL) {
12828             /* This is not a reentry */
12829             CONN_INC_REF(connp);
12830             CONN_INC_IOCTLREF(connp);
12831         } else {
12832             if (!(ipip->ipi_flags & IPI_MODOK)) {
12833                 mi_copy_done(q, mp, EINVAL);
12834                 return;
12835             }
12836         }
12837
12838         ip_process_ioctl(NULL, q, mp, ipip);
12839
12840     } else {
12841         mi_copyout(q, mp);
12842     }
12843     return;
12844
12845 case M_IOCNAK:
12846     /*
12847      * The only way we could get here is if a resolver didn't like
12848      * an IOCTL we sent it. This shouldn't happen.
12849      */
12850     (void) mi_strlog(q, 1, SL_ERROR|SL_TRACE,
12851         "ip_wput_nondata: unexpected M_IOCNAK, ioc_cmd 0x%x",
12852         ((struct iocblk *)mp->b_rptr->ioc_cmd);
12853     freemsg(mp);
12854     return;
12855 case M_IOCACK:
12856     /* /dev/ip shouldn't see this */
12857     goto nak;
12858 case M_FLUSH:
12859     if (*mp->b_rptr & FLUSHW)
12860         flushq(q, FLUSHALL);
12861     if (q->q_next) {
12862         putnext(q, mp);
12863         return;
12864     }
12865     if (*mp->b_rptr & FLUSHR) {

```

```

12866         *mp->b_rptr &= ~FLUSHW;
12867         greply(q, mp);
12868         return;
12869     }
12870     freemsg(mp);
12871     return;
12872 case M_CTL:
12873     break;
12874 case M_PROTO:
12875 case M_PCPROTO:
12876     /*
12877      * The only PROTO messages we expect are SNMP-related.
12878      */
12879     switch (((union T_primitives *)mp->b_rptr->type) {
12880     case T_SVR4_OPTMGMT_REQ:
12881         ip2dbg(("ip_wput_nondata: T_SVR4_OPTMGMT_REQ "
12882             "flags %x\n",
12883             ((struct T_optmgmt_req *)mp->b_rptr->MGMT_flags));
12884
12885         if (connp == NULL) {
12886             proto_str = "T_SVR4_OPTMGMT_REQ";
12887             goto protonak;
12888         }
12889
12890         /*
12891          * All Solaris components should pass a db_credp
12892          * for this TPI message, hence we ASSERT.
12893          * But in case there is some other M_PROTO that looks
12894          * like a TPI message sent by some other kernel
12895          * component, we check and return an error.
12896          */
12897         cr = msg_getcred(mp, NULL);
12898         ASSERT(cr != NULL);
12899         if (cr == NULL) {
12900             mp = mi_tpi_err_ack_alloc(mp, TSYSERR, EINVAL);
12901             if (mp != NULL)
12902                 greply(q, mp);
12903             return;
12904         }
12905
12906         if (!snmpcom_req(q, mp, ip_snmp_set, ip_snmp_get, cr)) {
12907             proto_str = "Bad SNMPCOM request?";
12908             goto protonak;
12909         }
12910         return;
12911     default:
12912         ipldbg(("ip_wput_nondata: dropping M_PROTO prim %u\n",
12913             (int)*(uint_t *)mp->b_rptr));
12914         freemsg(mp);
12915         return;
12916     }
12917     default:
12918         break;
12919 }
12920 if (q->q_next) {
12921     putnext(q, mp);
12922 } else
12923     freemsg(mp);
12924 return;
12925
12926 nak:
12927     iocp->ioc_error = EINVAL;
12928     mp->b_datap->db_type = M_IOCNAK;
12929     iocp->ioc_count = 0;
12930     greply(q, mp);
12931     return;

```

```

12933 protonak:
12934     cmn_err(CE_NOTE, "IP doesn't process %s as a module", proto_str);
12935     if ((mp = mi_tpi_err_ack_alloc(mp, TPROTO, EINVAL)) != NULL)
12936         greply(q, mp);
12937 }

12939 /*
12940  * Process IP options in an outbound packet.  Verify that the nexthop in a
12941  * strict source route is onlink.
12942  * Returns non-zero if something fails in which case an ICMP error has been
12943  * sent and mp freed.
12944  *
12945  * Assumes the ULP has called ip_message_options to move nexthop into ipha_dst.
12946  */
12947 int
12948 ip_output_options(mblk_t *mp, ipha_t *ipha, ip_xmit_attr_t *ixa, ill_t *ill)
12949 {
12950     ipoptp_t     opts;
12951     uchar_t     *opt;
12952     uint8_t     optval;
12953     uint8_t     optlen;
12954     ipaddr_t     dst;
12955     intptr_t     code = 0;
12956     ire_t        *ire;
12957     ip_stack_t   *ipst = ix->ixa_ipst;
12958     ip_rcv_attr_t iras;

12960     ip2dbg(("ip_output_options\n"));

12962     dst = ipha->ipha_dst;
12963     for (optval = ipoptp_first(&opts, ipha);
12964          optval != IPOPT_EOL;
12965          optval = ipoptp_next(&opts)) {
12966         opt = opts.ipoptp_cur;
12967         optlen = opts.ipoptp_len;
12968         ip2dbg(("ip_output_options: opt %d, len %d\n",
12969                optval, optlen));
12970         switch (optval) {
12971             uint32_t off;
12972         case IPOPT_SSRR:
12973         case IPOPT_LSRR:
12974             if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
12975                 ip1dbg(("
12976                    "ip_output_options: bad option offset\n"));
12977                 code = (char *)&opt[IPOPT_OLEN] -
12978                       (char *)ipha;
12979                 goto param_prob;
12980             }
12981             off = opt[IPOPT_OFFSET];
12982             ip1dbg(("ip_output_options: next hop 0x%x\n",
12983                    ntohl(dst)));
12984             /*
12985              * For strict: verify that dst is directly
12986              * reachable.
12987              */
12988             if (optval == IPOPT_SSRR) {
12989                 ire = ire_fhtable_lookup_v4(dst, 0, 0,
12990                        IRE_INTERFACE, NULL, ALL_ZONES,
12991                        ix->ixa_tsl,
12992                        MATCH_IRE_TYPE | MATCH_IRE_SECATTR, 0, ipst,
12993                        NULL);
12994                 if (ire == NULL) {
12995                     ip1dbg(("ip_output_options: SSRR not"
12996                            " directly reachable: 0x%x\n",
12997                            ntohl(dst)));

```

```

12998         goto bad_src_route;
12999     }
13000     ire_refrele(ire);
13001 }
13002 break;
13003 case IPOPT_RR:
13004     if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
13005         ip1dbg(("
13006            "ip_output_options: bad option offset\n"));
13007         code = (char *)&opt[IPOPT_OLEN] -
13008               (char *)ipha;
13009         goto param_prob;
13010     }
13011     break;
13012 case IPOPT_TS:
13013     /*
13014      * Verify that length >=5 and that there is either
13015      * room for another timestamp or that the overflow
13016      * counter is not maxed out.
13017      */
13018     code = (char *)&opt[IPOPT_OLEN] - (char *)ipha;
13019     if (optlen < IPOPT_MINLEN_IT) {
13020         goto param_prob;
13021     }
13022     if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
13023         ip1dbg(("
13024            "ip_output_options: bad option offset\n"));
13025         code = (char *)&opt[IPOPT_OFFSET] -
13026               (char *)ipha;
13027         goto param_prob;
13028     }
13029     switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
13030     case IPOPT_TS_TSONLY:
13031         off = IPOPT_TS_TIMELEN;
13032         break;
13033     case IPOPT_TS_TSANDADDR:
13034     case IPOPT_TS_PRESPEC:
13035     case IPOPT_TS_PRESPEC_RFC791:
13036         off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
13037         break;
13038     default:
13039         code = (char *)&opt[IPOPT_POS_OV_FLG] -
13040               (char *)ipha;
13041         goto param_prob;
13042     }
13043     if (opt[IPOPT_OFFSET] - 1 + off > optlen &&
13044         (opt[IPOPT_POS_OV_FLG] & 0xF0) == 0xF0) {
13045         /*
13046          * No room and the overflow counter is 15
13047          * already.
13048          */
13049         goto param_prob;
13050     }
13051     break;
13052 }
13053 }

13055     if ((opts.ipoptp_flags & IPOPTP_ERROR) == 0)
13056         return (0);

13058     ip1dbg(("ip_output_options: error processing IP options."));
13059     code = (char *)&opt[IPOPT_OFFSET] - (char *)ipha;

13061 param_prob:
13062     bzero(&iras, sizeof (iras));
13063     iras.ira_ill = iras.ira_rill = ill;

```

```

13064     iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
13065     iras.ira_rifindex = iras.ira_ruifindex;
13066     iras.ira_flags = IRAF_IS_IPV4;

13068     ip_drop_output("ip_output_options", mp, ill);
13069     icmp_param_problem(mp, (uint8_t)code, &iras);
13070     ASSERT(! (iras.ira_flags & IRAF_IPSEC_SECURE));
13071     return (-1);

13073 bad_src_route:
13074     bzero(&iras, sizeof (iras));
13075     iras.ira_ill = iras.ira_rill = ill;
13076     iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
13077     iras.ira_rifindex = iras.ira_ruifindex;
13078     iras.ira_flags = IRAF_IS_IPV4;

13080     ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
13081     icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, &iras);
13082     ASSERT(! (iras.ira_flags & IRAF_IPSEC_SECURE));
13083     return (-1);
13084 }

13086 /*
13087  * The maximum value of conn_drain_list_cnt is CONN_MAXDRAINCNT.
13088  * conn_drain_list_cnt can be changed by setting conn_drain_nthreads
13089  * thru /etc/system.
13090  */
13091 #define CONN_MAXDRAINCNT      64

13093 static void
13094 conn_drain_init(ip_stack_t *ipst)
13095 {
13096     int i, j;
13097     idl_tx_list_t *itl_tx;

13099     ipst->ips_conn_drain_list_cnt = conn_drain_nthreads;

13101     if ((ipst->ips_conn_drain_list_cnt == 0) ||
13102         (ipst->ips_conn_drain_list_cnt > CONN_MAXDRAINCNT)) {
13103         /*
13104          * Default value of the number of drainers is the
13105          * number of cpus, subject to maximum of 8 drainers.
13106          */
13107         if (boot_max_ncpus != -1)
13108             ipst->ips_conn_drain_list_cnt = MIN(boot_max_ncpus, 8);
13109         else
13110             ipst->ips_conn_drain_list_cnt = MIN(max_ncpus, 8);
13111     }

13113     ipst->ips_idl_tx_list =
13114         kmem_zalloc(TX_FANOUT_SIZE * sizeof (idl_tx_list_t), KM_SLEEP);
13115     for (i = 0; i < TX_FANOUT_SIZE; i++) {
13116         itl_tx = &ipst->ips_idl_tx_list[i];
13117         itl_tx->txl_drain_list =
13118             kmem_zalloc(ipst->ips_conn_drain_list_cnt *
13119                 sizeof (idl_t), KM_SLEEP);
13120         mutex_init(&itl_tx->txl_lock, NULL, MUTEX_DEFAULT, NULL);
13121         for (j = 0; j < ipst->ips_conn_drain_list_cnt; j++) {
13122             mutex_init(&itl_tx->txl_drain_list[j].idl_lock, NULL,
13123                 MUTEX_DEFAULT, NULL);
13124             itl_tx->txl_drain_list[j].idl_itl = itl_tx;
13125         }
13126     }
13127 }

13129 static void

```

```

13130 conn_drain_fini(ip_stack_t *ipst)
13131 {
13132     int i;
13133     idl_tx_list_t *itl_tx;

13135     for (i = 0; i < TX_FANOUT_SIZE; i++) {
13136         itl_tx = &ipst->ips_idl_tx_list[i];
13137         kmem_free(itl_tx->txl_drain_list,
13138             ipst->ips_conn_drain_list_cnt * sizeof (idl_t));
13139     }
13140     kmem_free(ipst->ips_idl_tx_list,
13141         TX_FANOUT_SIZE * sizeof (idl_tx_list_t));
13142     ipst->ips_idl_tx_list = NULL;
13143 }

13145 /*
13146  * Flow control has blocked us from proceeding. Insert the given conn in one
13147  * of the conn drain lists. When flow control is unblocked, either ip_wsrvt()
13148  * (STREAMS) or ill_flow_enable() (direct) will be called back, which in turn
13149  * will call conn_walk_drain(). See the flow control notes at the top of this
13150  * file for more details.
13151  */
13152 void
13153 conn_drain_insert(conn_t *connp, idl_tx_list_t *tx_list)
13154 {
13155     idl_t *idl = tx_list->txl_drain_list;
13156     uint_t index;
13157     ip_stack_t *ipst = connp->conn_netstack->netstack_ip;

13159     mutex_enter(&connp->conn_lock);
13160     if (connp->conn_state_flags & CONN_CLOSING) {
13161         /*
13162          * The conn is closing as a result of which CONN_CLOSING
13163          * is set. Return.
13164          */
13165         mutex_exit(&connp->conn_lock);
13166         return;
13167     } else if (connp->conn_idl == NULL) {
13168         /*
13169          * Assign the next drain list round robin. We dont' use
13170          * a lock, and thus it may not be strictly round robin.
13171          * Atomicity of load/stores is enough to make sure that
13172          * conn_drain_list_index is always within bounds.
13173          */
13174         index = tx_list->txl_drain_index;
13175         ASSERT(index < ipst->ips_conn_drain_list_cnt);
13176         connp->conn_idl = &tx_list->txl_drain_list[index];
13177         index++;
13178         if (index == ipst->ips_conn_drain_list_cnt)
13179             index = 0;
13180         tx_list->txl_drain_index = index;
13181     } else {
13182         ASSERT(connp->conn_idl->idl_itl == tx_list);
13183     }
13184     mutex_exit(&connp->conn_lock);

13186     idl = connp->conn_idl;
13187     mutex_enter(&idl->idl_lock);
13188     if ((connp->conn_drain_prev != NULL) ||
13189         (connp->conn_state_flags & CONN_CLOSING)) {
13190         /*
13191          * The conn is either already in the drain list or closing.
13192          * (We needed to check for CONN_CLOSING again since close can
13193          * sneak in between dropping conn_lock and acquiring idl_lock.)
13194          */
13195         mutex_exit(&idl->idl_lock);

```

```

13196         return;
13197     }
13199     /*
13200     * The conn is not in the drain list. Insert it at the
13201     * tail of the drain list. The drain list is circular
13202     * and doubly linked. idl_conn points to the 1st element
13203     * in the list.
13204     */
13205     if (idl->idl_conn == NULL) {
13206         idl->idl_conn = connp;
13207         connp->conn_drain_next = connp;
13208         connp->conn_drain_prev = connp;
13209     } else {
13210         conn_t *head = idl->idl_conn;
13212         connp->conn_drain_next = head;
13213         connp->conn_drain_prev = head->conn_drain_prev;
13214         head->conn_drain_prev->conn_drain_next = connp;
13215         head->conn_drain_prev = connp;
13216     }
13217     /*
13218     * For non streams based sockets assert flow control.
13219     */
13220     conn_setqfull(connp, NULL);
13221     mutex_exit(&idl->idl_lock);
13222 }
13224 static void
13225 conn_drain_remove(conn_t *connp)
13226 {
13227     idl_t *idl = connp->conn_idl;
13229     if (idl != NULL) {
13230         /*
13231         * Remove ourself from the drain list.
13232         */
13233         if (connp->conn_drain_next == connp) {
13234             /* Singleton in the list */
13235             ASSERT(connp->conn_drain_prev == connp);
13236             idl->idl_conn = NULL;
13237         } else {
13238             connp->conn_drain_prev->conn_drain_next =
13239                 connp->conn_drain_next;
13240             connp->conn_drain_next->conn_drain_prev =
13241                 connp->conn_drain_prev;
13242             if (idl->idl_conn == connp)
13243                 idl->idl_conn = connp->conn_drain_next;
13244         }
13246         /*
13247         * NOTE: because conn_idl is associated with a specific drain
13248         * list which in turn is tied to the index the TX ring
13249         * (txl_cookie) hashes to, and because the TX ring can change
13250         * over the lifetime of the conn_t, we must clear conn_idl so
13251         * a subsequent conn_drain_insert() will set conn_idl again
13252         * based on the latest txl_cookie.
13253         */
13254         connp->conn_idl = NULL;
13255     }
13256     connp->conn_drain_next = NULL;
13257     connp->conn_drain_prev = NULL;
13259     conn_clrqfull(connp, NULL);
13260     /*
13261     * For streams based sockets open up flow control.

```

```

13262     */
13263     if (!IPCL_IS_NONSTR(connp))
13264         enablelock(connp->conn_wq);
13265 }
13267 /*
13268 * This conn is closing, and we are called from ip_close. OR
13269 * this conn is draining because flow-control on the ill has been relieved.
13270 *
13271 * We must also need to remove conn's on this idl from the list, and also
13272 * inform the sockfs upcalls about the change in flow-control.
13273 */
13274 static void
13275 conn_drain(conn_t *connp, boolean_t closing)
13276 {
13277     idl_t *idl;
13278     conn_t *next_connp;
13280     /*
13281     * connp->conn_idl is stable at this point, and no lock is needed
13282     * to check it. If we are called from ip_close, close has already
13283     * set CONN_CLOSING, thus freezing the value of conn_idl, and
13284     * called us only because conn_idl is non-null. If we are called thru
13285     * service, conn_idl could be null, but it cannot change because
13286     * service is single-threaded per queue, and there cannot be another
13287     * instance of service trying to call conn_drain_insert on this conn
13288     * now.
13289     */
13290     ASSERT(!closing || connp == NULL || connp->conn_idl != NULL);
13292     /*
13293     * If the conn doesn't exist or is not on a drain list, bail.
13294     */
13295     if (connp == NULL || connp->conn_idl == NULL ||
13296         connp->conn_drain_prev == NULL) {
13297         return;
13298     }
13300     idl = connp->conn_idl;
13301     ASSERT(MUTEX_HELD(&idl->idl_lock));
13303     if (!closing) {
13304         next_connp = connp->conn_drain_next;
13305         while (next_connp != connp) {
13306             conn_t *delconnp = next_connp;
13308             next_connp = next_connp->conn_drain_next;
13309             conn_drain_remove(delconnp);
13310         }
13311         ASSERT(connp->conn_drain_next == idl->idl_conn);
13312     }
13313     conn_drain_remove(connp);
13314 }
13316 /*
13317 * Write service routine. Shared perimeter entry point.
13318 * The device queue's messages has fallen below the low water mark and STREAMS
13319 * has backenabled the ill_wq. Send sockfs notification about flow-control on
13320 * each waiting conn.
13321 */
13322 void
13323 ip_wsrv(queue_t *q)
13324 {
13325     ill_t *ill;
13327     ill = (ill_t *)q->q_ptr;

```

```

13328     if (ill->ill_state_flags == 0) {
13329         ip_stack_t *ipst = ill->ill_ipst;

13331         /*
13332          * The device flow control has opened up.
13333          * Walk through conn drain lists and qenable the
13334          * first conn in each list. This makes sense only
13335          * if the stream is fully plumbed and setup.
13336          * Hence the ill_state_flags check above.
13337          */
13338         ipldbg(("ip_wsrv: walking\n"));
13339         conn_walk_drain(ipst, &ipst->ips_idl_tx_list[0]);
13340         enableok(ill->ill_wq);
13341     }
13342 }

13344 /*
13345  * Callback to disable flow control in IP.
13346  */
13347  * This is a mac client callback added when the DLD_CAPAB_DIRECT capability
13348  * is enabled.
13349  */
13350  * When MAC_TX() is not able to send any more packets, dld sets its queue
13351  * to QFULL and enable the STREAMS flow control. Later, when the underlying
13352  * driver is able to continue to send packets, it calls mac_tx(ring_update())
13353  * function and wakes up corresponding mac worker threads, which in turn
13354  * calls this callback function, and disables flow control.
13355  */
13356 void
13357 ill_flow_enable(void *arg, ip_mac_tx_cookie_t cookie)
13358 {
13359     ill_t *ill = (ill_t *)arg;
13360     ip_stack_t *ipst = ill->ill_ipst;
13361     idl_tx_list_t *idl_txl;

13363     idl_txl = &ipst->ips_idl_tx_list[IDLHASHINDEX(cookie)];
13364     mutex_enter(&idl_txl->txl_lock);
13365     /* add code to to set a flag to indicate idl_txl is enabled */
13366     conn_walk_drain(ipst, idl_txl);
13367     mutex_exit(&idl_txl->txl_lock);
13368 }

13370 /*
13371  * Flow control has been relieved and STREAMS has backenabled us; drain
13372  * all the conn lists on 'tx_list'.
13373  */
13374 static void
13375 conn_walk_drain(ip_stack_t *ipst, idl_tx_list_t *tx_list)
13376 {
13377     int i;
13378     idl_t *idl;

13380     IP_STAT(ipst, ip_conn_walk_drain);

13382     for (i = 0; i < ipst->ips_conn_drain_list_cnt; i++) {
13383         idl = &tx_list->txl_drain_list[i];
13384         mutex_enter(&idl->idl_lock);
13385         conn_drain(idl->idl_conn, B_FALSE);
13386         mutex_exit(&idl->idl_lock);
13387     }
13388 }

13390 /*
13391  * Determine if the ill and multicast aspects of that packets
13392  * "matches" the conn.
13393  */

```

```

13394 boolean_t
13395 conn_wantpacket(conn_t *connp, ip_recv_attr_t *ira, ipha_t *ipha)
13396 {
13397     ill_t          *ill = ira->ira_rill;
13398     zoneid_t       zoneid = ira->ira_zoneid;
13399     uint_t         in_ifindex;
13400     ipaddr_t       dst, src;

13402     dst = ipha->ipha_dst;
13403     src = ipha->ipha_src;

13405     /*
13406      * conn_incoming_ifindex is set by IP_BOUND_IF which limits
13407      * unicast, broadcast and multicast reception to
13408      * conn_incoming_ifindex.
13409      * conn_wantpacket is called for unicast, broadcast and
13410      * multicast packets.
13411      */
13412     in_ifindex = connp->conn_incoming_ifindex;

13414     /* mpathd can bind to the under IPMP interface, which we allow */
13415     if (in_ifindex != 0 && in_ifindex != ill->ill_phyint->phyint_ifindex) {
13416         if (!IS_UNDER_IPMP(ill))
13417             return (B_FALSE);

13419         if (in_ifindex != ipmp_ill_get_ipmp_ifindex(ill))
13420             return (B_FALSE);
13421     }

13423     if (!IPCL_ZONE_MATCH(connp, zoneid))
13424         return (B_FALSE);

13426     if (!(ira->ira_flags & IRAF_MULTICAST))
13427         return (B_TRUE);

13429     if (connp->conn_multi_router) {
13430         /* multicast packet and multicast router socket: send up */
13431         return (B_TRUE);
13432     }

13434     if (ipha->ipha_protocol == IPPROTO_PIM ||
13435         ipha->ipha_protocol == IPPROTO_RSVP)
13436         return (B_TRUE);

13438     return (conn_hasmembers_ill_withsrc_v4(connp, dst, src, ira->ira_ill));
13439 }

13441 void
13442 conn_setqfull(conn_t *connp, boolean_t *flow_stopped)
13443 {
13444     if (IPCL_IS_NONSTR(connp)) {
13445         (*connp->conn_upcalls->su_txq_full)
13446             (connp->conn_upper_handle, B_TRUE);
13447         if (flow_stopped != NULL)
13448             *flow_stopped = B_TRUE;
13449     } else {
13450         queue_t *q = connp->conn_wq;

13452         ASSERT(q != NULL);
13453         if (!(q->q_flag & QFULL)) {
13454             mutex_enter(QLOCK(q));
13455             if (!(q->q_flag & QFULL)) {
13456                 /* still need to set QFULL */
13457                 q->q_flag |= QFULL;
13458                 /* set flow_stopped to true under QLOCK */
13459                 if (flow_stopped != NULL)

```

```

13460         *flow_stopped = B_TRUE;
13461         mutex_exit(QLOCK(q));
13462     } else {
13463         /* flow_stopped is left unchanged */
13464         mutex_exit(QLOCK(q));
13465     }
13466 }
13467 }
13468 }

13470 void
13471 conn_clrqfull(conn_t *connp, boolean_t *flow_stopped)
13472 {
13473     if (IPCL_IS_NONSTR(connp)) {
13474         (*connp->conn_upcalls->su_txq_full)
13475         (connp->conn_upper_handle, B_FALSE);
13476         if (flow_stopped != NULL)
13477             *flow_stopped = B_FALSE;
13478     } else {
13479         queue_t *q = connp->conn_wq;

13481         ASSERT(q != NULL);
13482         if (q->q_flag & QFULL) {
13483             mutex_enter(QLOCK(q));
13484             if (q->q_flag & QFULL) {
13485                 q->q_flag &= ~QFULL;
13486                 /* set flow_stopped to false under QLOCK */
13487                 if (flow_stopped != NULL)
13488                     *flow_stopped = B_FALSE;
13489                 mutex_exit(QLOCK(q));
13490                 if (q->q_flag & QWANTW)
13491                     qbackenable(q, 0);
13492             } else {
13493                 /* flow_stopped is left unchanged */
13494                 mutex_exit(QLOCK(q));
13495             }
13496         }

13499         mutex_enter(&connp->conn_lock);
13500         connp->conn_blocked = B_FALSE;
13501         mutex_exit(&connp->conn_lock);
13502     }

13504 /*
13505  * Return the length in bytes of the IPv4 headers (base header, label, and
13506  * other IP options) that will be needed based on the
13507  * ip_pkt_t structure passed by the caller.
13508  *
13509  * The returned length does not include the length of the upper level
13510  * protocol (ULP) header.
13511  * The caller needs to check that the length doesn't exceed the max for IPv4.
13512  */
13513 int
13514 ip_total_hdrs_len_v4(const ip_pkt_t *ipp)
13515 {
13516     int len;

13518     len = IP_SIMPLE_HDR_LENGTH;
13519     if (ipp->ipp_fields & IPPF_LABEL_V4) {
13520         ASSERT(ipp->ipp_label_len_v4 != 0);
13521         /* We need to round up here */
13522         len += (ipp->ipp_label_len_v4 + 3) & ~3;
13523     }

13525     if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {

```

```

13526         ASSERT(ipp->ipp_ipv4_options_len != 0);
13527         ASSERT((ipp->ipp_ipv4_options_len & 3) == 0);
13528         len += ipp->ipp_ipv4_options_len;
13529     }
13530     return (len);
13531 }

13533 /*
13534  * All-purpose routine to build an IPv4 header with options based
13535  * on the abstract ip_pkt_t.
13536  *
13537  * The caller has to set the source and destination address as well as
13538  * ipha_length. The caller has to massage any source route and compensate
13539  * for the ULP pseudo-header checksum due to the source route.
13540  */
13541 void
13542 ip_build_hdrs_v4(uchar_t *buf, uint_t buf_len, const ip_pkt_t *ipp,
13543                 uint8_t protocol)
13544 {
13545     ipha_t *ipha = (ipha_t *)buf;
13546     uint8_t *cp;

13548     /* Initialize IPv4 header */
13549     ipha->ipha_type_of_service = ipp->ipp_type_of_service;
13550     ipha->ipha_length = 0; /* Caller will set later */
13551     ipha->ipha_ident = 0;
13552     ipha->ipha_fragment_offset_and_flags = 0;
13553     ipha->ipha_ttl = ipp->ipp_unicast_hops;
13554     ipha->ipha_protocol = protocol;
13555     ipha->ipha_hdr_checksum = 0;

13557     if ((ipp->ipp_fields & IPPF_ADDR) &&
13558         IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr))
13559         ipha->ipha_src = ipp->ipp_addr_v4;

13561     cp = (uint8_t *)&ipha[1];
13562     if (ipp->ipp_fields & IPPF_LABEL_V4) {
13563         ASSERT(ipp->ipp_label_len_v4 != 0);
13564         bcopy(ipp->ipp_label_v4, cp, ipp->ipp_label_len_v4);
13565         cp += ipp->ipp_label_len_v4;
13566         /* We need to round up here */
13567         while ((uintptr_t)cp & 0x3) {
13568             *cp++ = IPOPT_NOP;
13569         }
13570     }

13572     if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
13573         ASSERT(ipp->ipp_ipv4_options_len != 0);
13574         ASSERT((ipp->ipp_ipv4_options_len & 3) == 0);
13575         bcopy(ipp->ipp_ipv4_options, cp, ipp->ipp_ipv4_options_len);
13576         cp += ipp->ipp_ipv4_options_len;
13577     }
13578     ipha->ipha_version_and_hdr_length =
13579         (uint8_t)((IP_VERSION << 4) + buf_len / 4);

13581     ASSERT((int)(cp - buf) == buf_len);
13582 }

13584 /* Allocate the private structure */
13585 static int
13586 ip_priv_alloc(void **bufp)
13587 {
13588     void *buf;

13590     if ((buf = kmem_alloc(sizeof(ip_priv_t), KM_NOSLEEP)) == NULL)
13591         return (ENOMEM);

```

```

13593     *bufp = buf;
13594     return (0);
13595 }

13597 /* Function to delete the private structure */
13598 void
13599 ip_priv_free(void *buf)
13600 {
13601     ASSERT(buf != NULL);
13602     kmem_free(buf, sizeof (ip_priv_t));
13603 }

13605 /*
13606  * The entry point for IPPF processing.
13607  * If the classifier (IPGPC_CLASSIFY) is not loaded and configured, the
13608  * routine just returns.
13609  *
13610  * When called, ip_process generates an ipp_packet_t structure
13611  * which holds the state information for this packet and invokes the
13612  * the classifier (via ipp_packet_process). The classification, depending on
13613  * configured filters, results in a list of actions for this packet. Invoking
13614  * an action may cause the packet to be dropped, in which case we return NULL.
13615  * proc indicates the callout position for
13616  * this packet and ill is the interface this packet arrived on or will leave
13617  * on (inbound and outbound resp.).
13618  *
13619  * We do the processing on the rill (mapped to the upper if ipmp), but MIB
13620  * on the ill corresponding to the destination IP address.
13621  */
13622 mblk_t *
13623 ip_process(ip_proc_t proc, mblk_t *mp, ill_t *rill, ill_t *ill)
13624 {
13625     ip_priv_t     *priv;
13626     ipp_action_id_t aid;
13627     int           rc = 0;
13628     ipp_packet_t  *pp;

13630     /* If the classifier is not loaded, return */
13631     if ((aid = ipp_action_lookup(IPGPC_CLASSIFY)) == IPP_ACTION_INVAL) {
13632         return (mp);
13633     }

13635     ASSERT(mp != NULL);

13637     /* Allocate the packet structure */
13638     rc = ipp_packet_alloc(&pp, "ip", aid);
13639     if (rc != 0)
13640         goto drop;

13642     /* Allocate the private structure */
13643     rc = ip_priv_alloc((void **)&priv);
13644     if (rc != 0) {
13645         ipp_packet_free(pp);
13646         goto drop;
13647     }
13648     priv->proc = proc;
13649     priv->ill_index = ill_get_upper_ifindex(rill);

13651     ipp_packet_set_private(pp, priv, ip_priv_free);
13652     ipp_packet_set_data(pp, mp);

13654     /* Invoke the classifier */
13655     rc = ipp_packet_process(&pp);
13656     if (pp != NULL) {
13657         mp = ipp_packet_get_data(pp);

```

```

13658         ipp_packet_free(pp);
13659         if (rc != 0)
13660             goto drop;
13661         return (mp);
13662     } else {
13663         /* No mp to trace in ip_drop_input/ip_drop_output */
13664         mp = NULL;
13665     }
13666 drop:
13667     if (proc == IPP_LOCAL_IN || proc == IPP_FWD_IN) {
13668         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
13669         ip_drop_input("ip_process", mp, ill);
13670     } else {
13671         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
13672         ip_drop_output("ip_process", mp, ill);
13673     }
13674     freemsg(mp);
13675     return (NULL);
13676 }

13678 /*
13679  * Propagate a multicast group membership operation (add/drop) on
13680  * all the interfaces crossed by the related multirt routes.
13681  * The call is considered successful if the operation succeeds
13682  * on at least one interface.
13683  *
13684  * This assumes that a set of IRE_HOST/RTF_MULTIRT has been created for the
13685  * multicast addresses with the ire argument being the first one.
13686  * We walk the bucket to find all the of those.
13687  *
13688  * Common to IPv4 and IPv6.
13689  */
13690 static int
13691 ip_multirt_apply_membership(int (*fn)(conn_t *, boolean_t,
13692     const in6_addr_t *, ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *),
13693     ire_t *ire, conn_t *connp, boolean_t checkonly, const in6_addr_t *v6group,
13694     mcast_record_t fmode, const in6_addr_t *v6src)
13695 {
13696     ire_t     *ire_gw;
13697     irb_t     *irb;
13698     int       ifindex;
13699     int       error = 0;
13700     int       result;
13701     ip_stack_t *ipst = ire->ire_ipst;
13702     ipaddr_t  group;
13703     boolean_t isv6;
13704     int       match_flags;

13706     if (IN6_IS_ADDR_V4MAPPED(v6group)) {
13707         IN6_V4MAPPED_TO_IPADDR(v6group, group);
13708         isv6 = B_FALSE;
13709     } else {
13710         isv6 = B_TRUE;
13711     }

13713     irb = ire->ire_bucket;
13714     ASSERT(irb != NULL);

13716     result = 0;
13717     irb_refhold(irb);
13718     for (; ire != NULL; ire = ire->ire_next) {
13719         if ((ire->ire_flags & RTF_MULTIRT) == 0)
13720             continue;

13722         /* We handle -ifp routes by matching on the ill if set */
13723         match_flags = MATCH_IRE_TYPE;

```

```

13724     if (ire->ire_ill != NULL)
13725         match_flags |= MATCH_IRE_ILL;

13727     if (isv6) {
13728         if (!IN6_ARE_ADDR_EQUAL(&ire->ire_addr_v6, v6group))
13729             continue;

13731         ire_gw = ire_fhtable_lookup_v6(&ire->ire_gateway_addr_v6,
13732             0, 0, IRE_INTERFACE, ire->ire_ill, ALL_ZONES, NULL,
13733             match_flags, 0, ipst, NULL);
13734     } else {
13735         if (ire->ire_addr != group)
13736             continue;

13738         ire_gw = ire_fhtable_lookup_v4(ire->ire_gateway_addr,
13739             0, 0, IRE_INTERFACE, ire->ire_ill, ALL_ZONES, NULL,
13740             match_flags, 0, ipst, NULL);
13741     }
13742     /* No interface route exists for the gateway; skip this ire. */
13743     if (ire_gw == NULL)
13744         continue;
13745     if (ire_gw->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) {
13746         ire_refrele(ire_gw);
13747         continue;
13748     }
13749     ASSERT(ire_gw->ire_ill != NULL);          /* IRE_INTERFACE */
13750     ifindex = ire_gw->ire_ill->ill_phyint->phyint_ifindex;

13752     /*
13753      * The operation is considered a success if
13754      * it succeeds at least once on any one interface.
13755      */
13756     error = fn(connp, checkonly, v6group, INADDR_ANY, ifindex,
13757         fmode, v6src);
13758     if (error == 0)
13759         result = CGTP_MCAST_SUCCESS;

13761     ire_refrele(ire_gw);
13762 }
13763 irb_refrele(irb);
13764 /*
13765  * Consider the call as successful if we succeeded on at least
13766  * one interface. Otherwise, return the last encountered error.
13767  */
13768 return (result == CGTP_MCAST_SUCCESS ? 0 : error);
13769 }

13771 /*
13772  * Return the expected CGTP hooks version number.
13773  */
13774 int
13775 ip_cgtp_filter_supported(void)
13776 {
13777     return (ip_cgtp_filter_rev);
13778 }

13780 /*
13781  * CGTP hooks can be registered by invoking this function.
13782  * Checks that the version number matches.
13783  */
13784 int
13785 ip_cgtp_filter_register(netstackid_t stackid, cgtp_filter_ops_t *ops)
13786 {
13787     netstack_t *ns;
13788     ip_stack_t *ipst;

```

```

13790     if (ops->cfo_filter_rev != CGTP_FILTER_REV)
13791         return (ENOTSUP);

13793     ns = netstack_find_by_stackid(stackid);
13794     if (ns == NULL)
13795         return (EINVAL);
13796     ipst = ns->netstack_ip;
13797     ASSERT(ipst != NULL);

13799     if (ipst->ips_ip_cgtp_filter_ops != NULL) {
13800         netstack_rele(ns);
13801         return (EALREADY);
13802     }

13804     ipst->ips_ip_cgtp_filter_ops = ops;

13806     ill_set_inputfn_all(ipst);

13808     netstack_rele(ns);
13809     return (0);
13810 }

13812 /*
13813  * CGTP hooks can be unregistered by invoking this function.
13814  * Returns ENXIO if there was no registration.
13815  * Returns EBUSY if the ndd variable has not been turned off.
13816  */
13817 int
13818 ip_cgtp_filter_unregister(netstackid_t stackid)
13819 {
13820     netstack_t *ns;
13821     ip_stack_t *ipst;

13823     ns = netstack_find_by_stackid(stackid);
13824     if (ns == NULL)
13825         return (EINVAL);
13826     ipst = ns->netstack_ip;
13827     ASSERT(ipst != NULL);

13829     if (ipst->ips_ip_cgtp_filter) {
13830         netstack_rele(ns);
13831         return (EBUSY);
13832     }

13834     if (ipst->ips_ip_cgtp_filter_ops == NULL) {
13835         netstack_rele(ns);
13836         return (ENXIO);
13837     }
13838     ipst->ips_ip_cgtp_filter_ops = NULL;

13840     ill_set_inputfn_all(ipst);

13842     netstack_rele(ns);
13843     return (0);
13844 }

13846 /*
13847  * Check whether there is a CGTP filter registration.
13848  * Returns non-zero if there is a registration, otherwise returns zero.
13849  * Note: returns zero if bad stackid.
13850  */
13851 int
13852 ip_cgtp_filter_is_registered(netstackid_t stackid)
13853 {
13854     netstack_t *ns;
13855     ip_stack_t *ipst;

```



```

13856     int ret;

13858     ns = netstack_find_by_stackid(stackid);
13859     if (ns == NULL)
13860         return (0);
13861     ipst = ns->netstack_ip;
13862     ASSERT(ipst != NULL);

13864     if (ipst->ips_ip_cgtp_filter_ops != NULL)
13865         ret = 1;
13866     else
13867         ret = 0;

13869     netstack_rele(ns);
13870     return (ret);
13871 }

13873 static int
13874 ip_squeue_switch(int val)
13875 {
13876     int rval;

13878     switch (val) {
13879     case IP_SQUEUE_ENTER_NODRAIN:
13880         rval = SQ_NODRAIN;
13881         break;
13882     case IP_SQUEUE_ENTER:
13883         rval = SQ_PROCESS;
13884         break;
13885     case IP_SQUEUE_FILL:
13886     default:
13887         rval = SQ_FILL;
13888         break;
13889     }
13890     return (rval);
13891 }

13893 static void *
13894 ip_kstat2_init(netstackid_t stackid, ip_stat_t *ip_statisticsp)
13895 {
13896     kstat_t *ksp;

13898     ip_stat_t template = {
13899         "ip_udp_fannorm",           KSTAT_DATA_UINT64 },
13900         "ip_udp_fanmb",           KSTAT_DATA_UINT64 },
13901         "ip_recv_pullup",         KSTAT_DATA_UINT64 },
13902         "ip_db_ref",              KSTAT_DATA_UINT64 },
13903         "ip_notaligned",          KSTAT_DATA_UINT64 },
13904         "ip_multimblk",           KSTAT_DATA_UINT64 },
13905         "ip_opt",                  KSTAT_DATA_UINT64 },
13906         "ipsec_proto_ahesp",      KSTAT_DATA_UINT64 },
13907         "ip_conn_flputbq",        KSTAT_DATA_UINT64 },
13908         "ip_conn_walk_drain",     KSTAT_DATA_UINT64 },
13909         "ip_out_sw_cksum",        KSTAT_DATA_UINT64 },
13910         "ip_out_sw_cksum_bytes",  KSTAT_DATA_UINT64 },
13911         "ip_in_sw_cksum",         KSTAT_DATA_UINT64 },
13912         "ip_ire_reclaim_calls",   KSTAT_DATA_UINT64 },
13913         "ip_ire_reclaim_deleted", KSTAT_DATA_UINT64 },
13914         "ip_nce_reclaim_calls",   KSTAT_DATA_UINT64 },
13915         "ip_nce_reclaim_deleted", KSTAT_DATA_UINT64 },
13916         "ip_dce_reclaim_calls",   KSTAT_DATA_UINT64 },
13917         "ip_dce_reclaim_deleted", KSTAT_DATA_UINT64 },
13918         "ip_tcp_in_full_hw_cksum_err", KSTAT_DATA_UINT64 },
13919         "ip_tcp_in_part_hw_cksum_err", KSTAT_DATA_UINT64 },
13920         "ip_tcp_in_sw_cksum_err",  KSTAT_DATA_UINT64 },
13921         "ip_udp_in_full_hw_cksum_err", KSTAT_DATA_UINT64 },

```

```

13922         "ip_udp_in_part_hw_cksum_err", KSTAT_DATA_UINT64 },
13923         "ip_udp_in_sw_cksum_err",     KSTAT_DATA_UINT64 },
13924         "conn_in_recvdstaddr",        KSTAT_DATA_UINT64 },
13925         "conn_in_recvopts",           KSTAT_DATA_UINT64 },
13926         "conn_in_recvif",             KSTAT_DATA_UINT64 },
13927         "conn_in_recvslla",           KSTAT_DATA_UINT64 },
13928         "conn_in_recvucred",          KSTAT_DATA_UINT64 },
13929         "conn_in_recvttl",            KSTAT_DATA_UINT64 },
13930         "conn_in_recvhopopts",        KSTAT_DATA_UINT64 },
13931         "conn_in_recvhoplmit",        KSTAT_DATA_UINT64 },
13932         "conn_in_recvdstopts",        KSTAT_DATA_UINT64 },
13933         "conn_in_recvrthrdstopts",    KSTAT_DATA_UINT64 },
13934         "conn_in_recvrthdr",          KSTAT_DATA_UINT64 },
13935         "conn_in_recvpktinfo",        KSTAT_DATA_UINT64 },
13936         "conn_in_recvtclass",         KSTAT_DATA_UINT64 },
13937         "conn_in_timestamp",          KSTAT_DATA_UINT64 },
13938     };

13940     ksp = kstat_create_netstack("ip", 0, "ipstat", "net",
13941         KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t),
13942         KSTAT_FLAG_VIRTUAL, stackid);

13944     if (ksp == NULL)
13945         return (NULL);

13947     bcopy(&template, ip_statisticsp, sizeof (template));
13948     ksp->ks_data = (void *)ip_statisticsp;
13949     ksp->ks_private = (void *) (uintptr_t)stackid;

13951     kstat_install(ksp);
13952     return (ksp);
13953 }

13955 static void
13956 ip_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
13957 {
13958     if (ksp != NULL) {
13959         ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
13960         kstat_delete_netstack(ksp, stackid);
13961     }
13962 }

13964 static void *
13965 ip_kstat_init(netstackid_t stackid, ip_stack_t *ipst)
13966 {
13967     kstat_t *ksp;

13969     ip_named_kstat_t template = {
13970         "forwarding",             KSTAT_DATA_UINT32, 0 },
13971         "defaultTTL",             KSTAT_DATA_UINT32, 0 },
13972         "inReceives",             KSTAT_DATA_UINT64, 0 },
13973         "inHdrErrors",            KSTAT_DATA_UINT32, 0 },
13974         "inAddrErrors",           KSTAT_DATA_UINT32, 0 },
13975         "forDatagrams",           KSTAT_DATA_UINT64, 0 },
13976         "inUnknownProtos",        KSTAT_DATA_UINT32, 0 },
13977         "inDiscards",             KSTAT_DATA_UINT32, 0 },
13978         "inDelivers",             KSTAT_DATA_UINT64, 0 },
13979         "outRequests",            KSTAT_DATA_UINT64, 0 },
13980         "outDiscards",            KSTAT_DATA_UINT32, 0 },
13981         "outNoRoutes",            KSTAT_DATA_UINT32, 0 },
13982         "reasmTimeout",           KSTAT_DATA_UINT32, 0 },
13983         "reasmReqds",             KSTAT_DATA_UINT32, 0 },
13984         "reasmOKs",               KSTAT_DATA_UINT32, 0 },
13985         "reasmFails",             KSTAT_DATA_UINT32, 0 },
13986         "fragOKs",                KSTAT_DATA_UINT32, 0 },
13987         "fragFails",              KSTAT_DATA_UINT32, 0 },

```

```

13988     {"fragCreates",      KSTAT_DATA_UINT32, 0 },
13989     {"addrEntrySize",   KSTAT_DATA_INT32, 0 },
13990     {"routeEntrySize",  KSTAT_DATA_INT32, 0 },
13991     {"netToMediaEntrySize", KSTAT_DATA_INT32, 0 },
13992     {"routingDiscards", KSTAT_DATA_UINT32, 0 },
13993     {"inErrs",          KSTAT_DATA_UINT32, 0 },
13994     {"noPorts",         KSTAT_DATA_UINT32, 0 },
13995     {"inChecksumErrs",  KSTAT_DATA_UINT32, 0 },
13996     {"reasmDuplicates", KSTAT_DATA_UINT32, 0 },
13997     {"reasmPartDups",   KSTAT_DATA_UINT32, 0 },
13998     {"forwProhibits",   KSTAT_DATA_UINT32, 0 },
13999     {"udpInChecksumErrs", KSTAT_DATA_UINT32, 0 },
14000     {"udpInOverflows",  KSTAT_DATA_UINT32, 0 },
14001     {"rawipInOverflows", KSTAT_DATA_UINT32, 0 },
14002     {"ipsecInSucceeded", KSTAT_DATA_UINT32, 0 },
14003     {"ipsecInFailed",   KSTAT_DATA_INT32, 0 },
14004     {"memberEntrySize", KSTAT_DATA_INT32, 0 },
14005     {"inIPv6",          KSTAT_DATA_UINT32, 0 },
14006     {"outIPv6",         KSTAT_DATA_UINT32, 0 },
14007     {"outSwitchIPv6",   KSTAT_DATA_UINT32, 0 },
14008 };

14010 ksp = kstat_create_netstack("ip", 0, "ip", "mib2", KSTAT_TYPE_NAMED,
14011     NUM_OF_FIELDS(ip_named_kstat_t), 0, stackid);
14012 if (ksp == NULL || ksp->ks_data == NULL)
14013     return (NULL);

14015 template.forwarding.value.ui32 = WE_ARE_FORWARDING(ipst) ? 1:2;
14016 template.defaultTTL.value.ui32 = (uint32_t)ipst->ips_ip_def_ttl;
14017 template.reasmTimeout.value.ui32 = ipst->ips_ip_reassembly_timeout;
14018 template.addrEntrySize.value.i32 = sizeof (mib2_ipAddrEntry_t);
14019 template.routeEntrySize.value.i32 = sizeof (mib2_ipRouteEntry_t);

14021 template.netToMediaEntrySize.value.i32 =
14022     sizeof (mib2_ipNetToMediaEntry_t);

14024 template.memberEntrySize.value.i32 = sizeof (ip6_member_t);

14026 bcopy(&template, ksp->ks_data, sizeof (template));
14027 ksp->ks_update = ip_kstat_update;
14028 ksp->ks_private = (void *) (uintptr_t) stackid;

14030 kstat_install(ksp);
14031 return (ksp);
14032 }

14034 static void
14035 ip_kstat_fini(netstackid_t stackid, kstat_t *ksp)
14036 {
14037     if (ksp != NULL) {
14038         ASSERT(stackid == (netstackid_t) (uintptr_t) ksp->ks_private);
14039         kstat_delete_netstack(ksp, stackid);
14040     }
14041 }

14043 static int
14044 ip_kstat_update(kstat_t *kp, int rw)
14045 {
14046     ip_named_kstat_t *ipkp;
14047     mib2_ipIfStatsEntry_t ipmib;
14048     ill_walk_context_t ctx;
14049     ill_t *ill;
14050     netstackid_t stackid = (zoneid_t) (uintptr_t) kp->ks_private;
14051     netstack_t *ns;
14052     ip_stack_t *ipst;

```

```

14054     if (kp == NULL || kp->ks_data == NULL)
14055         return (EIO);

14057     if (rw == KSTAT_WRITE)
14058         return (EACCES);

14060     ns = netstack_find_by_stackid(stackid);
14061     if (ns == NULL)
14062         return (-1);
14063     ipst = ns->netstack_ip;
14064     if (ipst == NULL) {
14065         netstack_rele(ns);
14066         return (-1);
14067     }
14068     ipkp = (ip_named_kstat_t *) kp->ks_data;

14070     bcopy(&ipst->ips_ip_mib, &ipmib, sizeof (ipmib));
14071     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
14072     ill = ILL_START_WALK_V4(&ctx, ipst);
14073     for (; ill != NULL; ill = ill_next(&ctx, ill))
14074         ip_mib2_add_ip_stats(&ipmib, ill->ill_ip_mib);
14075     rw_exit(&ipst->ips_ill_g_lock);

14077     ipkp->forwarding.value.ui32 = ipmib.ipIfStatsForwarding;
14078     ipkp->defaultTTL.value.ui32 = ipmib.ipIfStatsDefaultTTL;
14079     ipkp->inReceives.value.ui64 = ipmib.ipIfStatsHCInReceives;
14080     ipkp->inHdrErrors.value.ui32 = ipmib.ipIfStatsInHdrErrors;
14081     ipkp->inAddrErrors.value.ui32 = ipmib.ipIfStatsInAddrErrors;
14082     ipkp->forwDatagrams.value.ui64 = ipmib.ipIfStatsHCOUForwDatagrams;
14083     ipkp->inUnknownProtos.value.ui32 = ipmib.ipIfStatsInUnknownProtos;
14084     ipkp->inDiscards.value.ui32 = ipmib.ipIfStatsInDiscards;
14085     ipkp->inDelivers.value.ui64 = ipmib.ipIfStatsHCInDelivers;
14086     ipkp->outRequests.value.ui64 = ipmib.ipIfStatsOutRequests;
14087     ipkp->outDiscards.value.ui32 = ipmib.ipIfStatsOutDiscards;
14088     ipkp->outNoRoutes.value.ui32 = ipmib.ipIfStatsOutNoRoutes;
14089     ipkp->reasmTimeout.value.ui32 = ipst->ips_ip_reassembly_timeout;
14090     ipkp->reasmReqds.value.ui32 = ipmib.ipIfStatsReasmReqds;
14091     ipkp->reasmOKs.value.ui32 = ipmib.ipIfStatsReasmOKs;
14092     ipkp->reasmFails.value.ui32 = ipmib.ipIfStatsReasmFails;
14093     ipkp->fragOKs.value.ui32 = ipmib.ipIfStatsOutFragOKs;
14094     ipkp->fragFails.value.ui32 = ipmib.ipIfStatsOutFragFails;
14095     ipkp->fragCreates.value.ui32 = ipmib.ipIfStatsOutFragCreates;

14097     ipkp->routingDiscards.value.ui32 = 0;
14098     ipkp->inErrs.value.ui32 = ipmib.tcpIfStatsInErrs;
14099     ipkp->noPorts.value.ui32 = ipmib.udpIfStatsNoPorts;
14100     ipkp->inChecksumErrs.value.ui32 = ipmib.ipIfStatsInChecksumErrs;
14101     ipkp->reasmDuplicates.value.ui32 = ipmib.ipIfStatsReasmDuplicates;
14102     ipkp->reasmPartDups.value.ui32 = ipmib.ipIfStatsReasmPartDups;
14103     ipkp->forwProhibits.value.ui32 = ipmib.ipIfStatsForwProhibits;
14104     ipkp->udpInChecksumErrs.value.ui32 = ipmib.udpIfStatsInChecksumErrs;
14105     ipkp->udpInOverflows.value.ui32 = ipmib.udpIfStatsInOverflows;
14106     ipkp->rawipInOverflows.value.ui32 = ipmib.rawipIfStatsInOverflows;
14107     ipkp->ipsecInSucceeded.value.ui32 = ipmib.ipsecIfStatsInSucceeded;
14108     ipkp->ipsecInFailed.value.i32 = ipmib.ipsecIfStatsInFailed;

14110     ipkp->inIPv6.value.ui32 = ipmib.ipIfStatsInWrongIPVersion;
14111     ipkp->outIPv6.value.ui32 = ipmib.ipIfStatsOutWrongIPVersion;
14112     ipkp->outSwitchIPv6.value.ui32 = ipmib.ipIfStatsOutSwitchIPVersion;

14114     netstack_rele(ns);

14116     return (0);
14117 }

14119 static void *

```

```

14120 icmp_kstat_init(netstackid_t stackid)
14121 {
14122     kstat_t *ksp;

14124     icmp_named_kstat_t template = {
14125         "inMsgs",          KSTAT_DATA_UINT32 },
14126         "inErrors",       KSTAT_DATA_UINT32 },
14127         "inDestUnreachs", KSTAT_DATA_UINT32 },
14128         "inTimeExcds",   KSTAT_DATA_UINT32 },
14129         "inParmProbs",   KSTAT_DATA_UINT32 },
14130         "inSrcQuenchs",  KSTAT_DATA_UINT32 },
14131         "inRedirects",   KSTAT_DATA_UINT32 },
14132         "inEchos",       KSTAT_DATA_UINT32 },
14133         "inEchoReps",    KSTAT_DATA_UINT32 },
14134         "inTimestamps",  KSTAT_DATA_UINT32 },
14135         "inTimestampReps", KSTAT_DATA_UINT32 },
14136         "inAddrMasks",   KSTAT_DATA_UINT32 },
14137         "inAddrMaskReps", KSTAT_DATA_UINT32 },
14138         "outMsgs",        KSTAT_DATA_UINT32 },
14139         "outErrors",     KSTAT_DATA_UINT32 },
14140         "outDestUnreachs", KSTAT_DATA_UINT32 },
14141         "outTimeExcds",  KSTAT_DATA_UINT32 },
14142         "outParmProbs",  KSTAT_DATA_UINT32 },
14143         "outSrcQuenchs", KSTAT_DATA_UINT32 },
14144         "outRedirects",  KSTAT_DATA_UINT32 },
14145         "outEchos",      KSTAT_DATA_UINT32 },
14146         "outEchoReps",   KSTAT_DATA_UINT32 },
14147         "outTimestamps", KSTAT_DATA_UINT32 },
14148         "outTimestampReps", KSTAT_DATA_UINT32 },
14149         "outAddrMasks",  KSTAT_DATA_UINT32 },
14150         "outAddrMaskReps", KSTAT_DATA_UINT32 },
14151         "inChksumErrs",  KSTAT_DATA_UINT32 },
14152         "inUnknowns",    KSTAT_DATA_UINT32 },
14153         "inFragNeeded",  KSTAT_DATA_UINT32 },
14154         "outFragNeeded", KSTAT_DATA_UINT32 },
14155         "outDrops",      KSTAT_DATA_UINT32 },
14156         "inOverflows",   KSTAT_DATA_UINT32 },
14157         "inBadRedirects", KSTAT_DATA_UINT32 },
14158     };

14160     ksp = kstat_create_netstack("ip", 0, "icmp", "mib2", KSTAT_TYPE_NAMED,
14161         NUM_OF_FIELDS(icmp_named_kstat_t), 0, stackid);
14162     if (ksp == NULL || ksp->ks_data == NULL)
14163         return (NULL);

14165     bcopy(&template, ksp->ks_data, sizeof (template));

14167     ksp->ks_update = icmp_kstat_update;
14168     ksp->ks_private = (void *) (uintptr_t) stackid;

14170     kstat_install(ksp);
14171     return (ksp);
14172 }

14174 static void
14175 icmp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
14176 {
14177     if (ksp != NULL) {
14178         ASSERT(stackid == (netstackid_t) (uintptr_t) ksp->ks_private);
14179         kstat_delete_netstack(ksp, stackid);
14180     }
14181 }

14183 static int
14184 icmp_kstat_update(kstat_t *kp, int rw)
14185 {

```

```

14186     icmp_named_kstat_t *icmpkp;
14187     netstackid_t stackid = (zoneid_t) (uintptr_t) kp->ks_private;
14188     netstack_t *ns;
14189     ip_stack_t *ipst;

14191     if ((kp == NULL) || (kp->ks_data == NULL))
14192         return (EIO);

14194     if (rw == KSTAT_WRITE)
14195         return (EACCES);

14197     ns = netstack_find_by_stackid(stackid);
14198     if (ns == NULL)
14199         return (-1);
14200     ipst = ns->netstack_ip;
14201     if (ipst == NULL) {
14202         netstack_rele(ns);
14203         return (-1);
14204     }
14205     icmpkp = (icmp_named_kstat_t *) kp->ks_data;

14207     icmpkp->inMsgs.value.ui32 = ipst->ips_icmp_mib.icmpInMsgs;
14208     icmpkp->inErrors.value.ui32 = ipst->ips_icmp_mib.icmpInErrors;
14209     icmpkp->inDestUnreachs.value.ui32 =
14210         ipst->ips_icmp_mib.icmpInDestUnreachs;
14211     icmpkp->inTimeExcds.value.ui32 = ipst->ips_icmp_mib.icmpInTimeExcds;
14212     icmpkp->inParmProbs.value.ui32 = ipst->ips_icmp_mib.icmpInParmProbs;
14213     icmpkp->inSrcQuenchs.value.ui32 = ipst->ips_icmp_mib.icmpInSrcQuenchs;
14214     icmpkp->inRedirects.value.ui32 = ipst->ips_icmp_mib.icmpInRedirects;
14215     icmpkp->inEchos.value.ui32 = ipst->ips_icmp_mib.icmpInEchos;
14216     icmpkp->inEchoReps.value.ui32 = ipst->ips_icmp_mib.icmpInEchoReps;
14217     icmpkp->inTimestamps.value.ui32 = ipst->ips_icmp_mib.icmpInTimestamps;
14218     icmpkp->inTimestampReps.value.ui32 =
14219         ipst->ips_icmp_mib.icmpInTimestampReps;
14220     icmpkp->inAddrMasks.value.ui32 = ipst->ips_icmp_mib.icmpInAddrMasks;
14221     icmpkp->inAddrMaskReps.value.ui32 =
14222         ipst->ips_icmp_mib.icmpInAddrMaskReps;
14223     icmpkp->outMsgs.value.ui32 = ipst->ips_icmp_mib.icmpOutMsgs;
14224     icmpkp->outErrors.value.ui32 = ipst->ips_icmp_mib.icmpOutErrors;
14225     icmpkp->outDestUnreachs.value.ui32 =
14226         ipst->ips_icmp_mib.icmpOutDestUnreachs;
14227     icmpkp->outTimeExcds.value.ui32 = ipst->ips_icmp_mib.icmpOutTimeExcds;
14228     icmpkp->outParmProbs.value.ui32 = ipst->ips_icmp_mib.icmpOutParmProbs;
14229     icmpkp->outSrcQuenchs.value.ui32 =
14230         ipst->ips_icmp_mib.icmpOutSrcQuenchs;
14231     icmpkp->outRedirects.value.ui32 = ipst->ips_icmp_mib.icmpOutRedirects;
14232     icmpkp->outEchos.value.ui32 = ipst->ips_icmp_mib.icmpOutEchos;
14233     icmpkp->outEchoReps.value.ui32 = ipst->ips_icmp_mib.icmpOutEchoReps;
14234     icmpkp->outTimestamps.value.ui32 =
14235         ipst->ips_icmp_mib.icmpOutTimestamps;
14236     icmpkp->outTimestampReps.value.ui32 =
14237         ipst->ips_icmp_mib.icmpOutTimestampReps;
14238     icmpkp->outAddrMasks.value.ui32 =
14239         ipst->ips_icmp_mib.icmpOutAddrMasks;
14240     icmpkp->outAddrMaskReps.value.ui32 =
14241         ipst->ips_icmp_mib.icmpOutAddrMaskReps;
14242     icmpkp->inChksumErrs.value.ui32 = ipst->ips_icmp_mib.icmpInChksumErrs;
14243     icmpkp->inUnknowns.value.ui32 = ipst->ips_icmp_mib.icmpInUnknowns;
14244     icmpkp->inFragNeeded.value.ui32 = ipst->ips_icmp_mib.icmpInFragNeeded;
14245     icmpkp->outFragNeeded.value.ui32 =
14246         ipst->ips_icmp_mib.icmpOutFragNeeded;
14247     icmpkp->outDrops.value.ui32 = ipst->ips_icmp_mib.icmpOutDrops;
14248     icmpkp->inOverflows.value.ui32 = ipst->ips_icmp_mib.icmpInOverflows;
14249     icmpkp->inBadRedirects.value.ui32 =
14250         ipst->ips_icmp_mib.icmpInBadRedirects;

```

```

14252     netstack_rele(ns);
14253     return (0);
14254 }

14256 /*
14257  * This is the fanout function for raw socket opened for SCTP. Note
14258  * that it is called after SCTP checks that there is no socket which
14259  * wants a packet. Then before SCTP handles this out of the blue packet,
14260  * this function is called to see if there is any raw socket for SCTP.
14261  * If there is and it is bound to the correct address, the packet will
14262  * be sent to that socket. Note that only one raw socket can be bound to
14263  * a port. This is assured in ipcl_sctp_hash_insert();
14264  */
14265 void
14266 ip_fanout_sctp_raw(mblk_t *mp, ipha_t *ipha, ip6_t *ip6h, uint32_t ports,
14267     ip_rcv_attr_t *ira)
14268 {
14269     conn_t      *connp;
14270     queue_t     *rq;
14271     boolean_t   secure;
14272     ill_t       *ill = ira->ira_ill;
14273     ip_stack_t  *ipst = ill->ill_ipst;
14274     ipsec_stack_t *ipss = ipst->ips_netstack->netstack_ipsec;
14275     sctp_stack_t *sctps = ipst->ips_netstack->netstack_sctp;
14276     iaflags_t   iraflags = ira->ira_flags;
14277     ill_t       *rill = ira->ira_rill;

14279     secure = iraflags & IRAF_IPSEC_SECURE;

14281     connp = ipcl_classify_raw(mp, IPPROTO_SCTP, ports, ipha, ip6h,
14282     ira, ipst);
14283     if (connp == NULL) {
14284         /*
14285          * Although raw sctp is not summed, OOB chunks must be.
14286          * Drop the packet here if the sctp checksum failed.
14287          */
14288         if (iraflags & IRAF_SCTP_CSUM_ERR) {
14289             SCTPS_BUMP_MIB(sctps, sctpChecksumError);
14290             freemsg(mp);
14291             return;
14292         }
14293         ira->ira_ill = ira->ira_rill = NULL;
14294         sctp_oob_input(mp, ira, ipst);
14295         ira->ira_ill = ill;
14296         ira->ira_rill = rill;
14297         return;
14298     }
14299     rq = connp->conn_rq;
14300     if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrl : !canputnext(rq)) {
14301         CONN_DEC_REF(connp);
14302         BUMP_MIB(ill->ill_ip_mib, rawipIfStatsInOverflows);
14303         freemsg(mp);
14304         return;
14305     }
14306     if (((iraflags & IRAF_IS_IPV4) ?
14307         CONN_INBOUND_POLICY_PRESENT(connp, ipss) :
14308         CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)) ||
14309         secure) {
14310         mp = ipsec_check_inbound_policy(mp, connp, ipha,
14311         ip6h, ira);
14312         if (mp == NULL) {
14313             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
14314             /* Note that mp is NULL */
14315             ip_drop_input("ipIfStatsInDiscards", mp, ill);
14316             CONN_DEC_REF(connp);
14317             return;

```

```

14318     }
14319 }

14321     if (iraflags & IRAF_ICMP_ERROR) {
14322         (connp->conn_recvicmp)(connp, mp, NULL, ira);
14323     } else {
14324         ill_t *rill = ira->ira_rill;

14326         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
14327         /* This is the SOCK_RAW, IPPROTO_SCTP case. */
14328         ira->ira_ill = ira->ira_rill = NULL;
14329         (connp->conn_rcv)(connp, mp, NULL, ira);
14330         ira->ira_ill = ill;
14331         ira->ira_rill = rill;
14332     }
14333     CONN_DEC_REF(connp);
14334 }

14336 /*
14337  * Free a packet that has the link-layer dl_unitdata_req_t or fast-path
14338  * header before the ip payload.
14339  */
14340 static void
14341 ip_xmit_flowctl_drop(ill_t *ill, mblk_t *mp, boolean_t is_fp_mp, int fp_mp_len)
14342 {
14343     int len = (mp->b_wptr - mp->b_rptr);
14344     mblk_t *ip_mp;

14346     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
14347     if (is_fp_mp || len != fp_mp_len) {
14348         if (len > fp_mp_len) {
14349             /*
14350              * fastpath header and ip header in the first mblk
14351              */
14352             mp->b_rptr += fp_mp_len;
14353         } else {
14354             /*
14355              * ip_xmit_attach_llhdr had to prepend an mblk to
14356              * attach the fastpath header before ip header.
14357              */
14358             ip_mp = mp->b_cont;
14359             freeb(mp);
14360             mp = ip_mp;
14361             mp->b_rptr += (fp_mp_len - len);
14362         }
14363     } else {
14364         ip_mp = mp->b_cont;
14365         freeb(mp);
14366         mp = ip_mp;
14367     }
14368     ip_drop_output("ipIfStatsOutDiscards - flow ctl", mp, ill);
14369     freemsg(mp);
14370 }

14372 /*
14373  * Normal post fragmentation function.
14374  *
14375  * Send a packet using the passed in nce. This handles both IPv4 and IPv6
14376  * using the same state machine.
14377  *
14378  * We return an error on failure. In particular we return EWOULDBLOCK
14379  * when the driver flow controls. In that case this ensures that ip_wsrsv runs
14380  * (currently by canputnext failure resulting in backenabling from GLD.)
14381  * This allows the callers of conn_ip_output() to use EWOULDBLOCK as an
14382  * indication that they can flow control until ip_wsrsv() tells then to restart.
14383  */

```

```

14384 * If the nce passed by caller is incomplete, this function
14385 * queues the packet and if necessary, sends ARP request and bails.
14386 * If the Neighbor Cache passed is fully resolved, we simply prepend
14387 * the link-layer header to the packet, do ipsec hw acceleration
14388 * work if necessary, and send the packet out on the wire.
14389 */
14390 /* ARGSUSED6 */
14391 int
14392 ip_xmit(mblk_t *mp, nce_t *nce, iaflags_t iaflags, uint_t pkt_len,
14393        uint32_t xmit_hint, zoneid_t szone, zoneid_t nolzid, uintptr_t *ixacookie)
14394 {
14395     queue_t      *wq;
14396     ill_t        *ill = nce->nce_ill;
14397     ip_stack_t   *ipst = ill->ill_ipst;
14398     uint64_t     delta;
14399     boolean_t    isv6 = ill->ill_isv6;
14400     boolean_t    fp_mp;
14401     ncec_t       *ncec = nce->nce_common;
14402     int64_t      now = LBOLT_FASTPATH64;
14403     boolean_t    is_probe;

14405     DTRACE_PROBE1(ip_xmit, nce_t *, nce);

14407     ASSERT(mp != NULL);
14408     ASSERT(mp->b_datap->db_type == M_DATA);
14409     ASSERT(pkt_len == msgdsize(mp));

14411     /*
14412      * If we have already been here and are coming back after ARP/ND.
14413      * the IXAF_NO_TRACE flag is set. We skip FW_HOOKS, DTRACE and ipobs
14414      * in that case since they have seen the packet when it came here
14415      * the first time.
14416      */
14417     if (iaflags & IXAF_NO_TRACE)
14418         goto sendit;

14420     if (iaflags & IXAF_IS_IPV4) {
14421         ipha_t *ipha = (ipha_t *)mp->b_rptr;

14423         ASSERT(!isv6);
14424         ASSERT(pkt_len == ntohs(((ipha_t *)mp->b_rptr->ipha_length));
14425         if (HOOKS4_INTERESTED_PHYSICAL_OUT(ipst) &&
14426             !(iaflags & IXAF_NO_PFHOOK)) {
14427             int error;

14429             FW_HOOKS(ipst->ips_ip4_physical_out_event,
14430                    ipst->ips_ipv4firewall_physical_out,
14431                    NULL, ill, ipha, mp, mp, 0, ipst, error);
14432             DTRACE_PROBE1(ip4_physical_out_end,
14433                    mblk_t *, mp);
14434             if (mp == NULL)
14435                 return (error);

14437             /* The length could have changed */
14438             pkt_len = msgdsize(mp);
14439         }
14440         if (ipst->ips_ip4_observe.he_interested) {
14441             /*
14442              * Note that for TX the zoneid is the sending
14443              * zone, whether or not MLP is in play.
14444              * Since the szone argument is the IP zoneid (i.e.,
14445              * zero for exclusive-IP zones) and ipobs wants
14446              * the system zoneid, we map it here.
14447              */
14448             szone = IP_REAL_ZONEID(szone, ipst);

```

```

14450     /*
14451      * On the outbound path the destination zone will be
14452      * unknown as we're sending this packet out on the
14453      * wire.
14454      */
14455     ipobs_hook(mp, IPOBS_HOOK_OUTBOUND, szone, ALL_ZONES,
14456              ill, ipst);
14457 }
14458 DTRACE_IP7(send, mblk_t *, mp, conn_t *, NULL,
14459            void_ip_t *, ipha, __dtrace_ipsr_ill_t *, ill,
14460            ipha_t *, ipha, ip6_t *, NULL, int, 0);
14461 } else {
14462     ip6_t *ip6h = (ip6_t *)mp->b_rptr;

14464     ASSERT(isv6);
14465     ASSERT(pkt_len ==
14466            ntohs(((ip6_t *)mp->b_rptr->ip6_plen) + IPV6_HDR_LEN);
14467     if (HOOKS6_INTERESTED_PHYSICAL_OUT(ipst) &&
14468         !(iaflags & IXAF_NO_PFHOOK)) {
14469         int error;

14471         FW_HOOKS6(ipst->ips_ip6_physical_out_event,
14472                ipst->ips_ipv6firewall_physical_out,
14473                NULL, ill, ip6h, mp, mp, 0, ipst, error);
14474         DTRACE_PROBE1(ip6_physical_out_end,
14475                mblk_t *, mp);
14476         if (mp == NULL)
14477             return (error);

14479         /* The length could have changed */
14480         pkt_len = msgdsize(mp);
14481     }
14482     if (ipst->ips_ip6_observe.he_interested) {
14483         /* See above */
14484         szone = IP_REAL_ZONEID(szone, ipst);

14486         ipobs_hook(mp, IPOBS_HOOK_OUTBOUND, szone, ALL_ZONES,
14487                  ill, ipst);
14488     }
14489     DTRACE_IP7(send, mblk_t *, mp, conn_t *, NULL,
14490            void_ip_t *, ip6h, __dtrace_ipsr_ill_t *, ill,
14491            ipha_t *, NULL, ip6_t *, ip6h, int, 0);
14492 }

14494 sendit:
14495     /*
14496      * We check the state without a lock because the state can never
14497      * move "backwards" to initial or incomplete.
14498      */
14499     switch (ncec->ncec_state) {
14500     case ND_REACHABLE:
14501     case ND_STALE:
14502     case ND_DELAY:
14503     case ND_PROBE:
14504         mp = ip_xmit_attach_llhdr(mp, nce);
14505         if (mp == NULL) {
14506             /*
14507              * ip_xmit_attach_llhdr has increased
14508              * ipIfStatsOutDiscards and called ip_drop_output()
14509              */
14510             return (ENOBUFS);
14511         }
14512     /*
14513      * check if nce fastpath completed and we tagged on a
14514      * copy of nce_fp_mp in ip_xmit_attach_llhdr().
14515      */

```

```

14516         fp_mp = (mp->b_datap->db_type == M_DATA);
14518
14518         if (fp_mp &&
14519             (ill->ill_capabilities & ILL_CAPAB_DLD_DIRECT)) {
14520             ill_dld_direct_t *idd;
14522
14522             idd = &ill->ill_dld_capab->idc_direct;
14523             /*
14524              * Send the packet directly to DLD, where it
14525              * may be queued depending on the availability
14526              * of transmit resources at the media layer.
14527              * Return value should be taken into
14528              * account and flow control the TCP.
14529              */
14530             BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCOutTransmits);
14531             UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCOutOctets,
14532                      pkt_len);
14534
14534             if (ixaflags & IXAF_NO_DEV_FLOW_CTL) {
14535                 (void) idd->idd_tx_df(idd->idd_tx_dh, mp,
14536                                     (uintptr_t)xmit_hint, IP_DROP_ON_NO_DESC);
14537             } else {
14538                 uintptr_t cookie;
14540
14540                 if ((cookie = idd->idd_tx_df(idd->idd_tx_dh,
14541                                             mp, (uintptr_t)xmit_hint, 0)) != 0) {
14542                     if (ixacookie != NULL)
14543                         *ixacookie = cookie;
14544                     return (EWOULDBLOCK);
14545                 }
14546             }
14547         } else {
14548             wq = ill->ill_wq;
14550
14550             if (!(ixaflags & IXAF_NO_DEV_FLOW_CTL) &&
14551                 !canputnext(wq)) {
14552                 if (ixacookie != NULL)
14553                     *ixacookie = 0;
14554                 ip_xmit_flowctl_drop(ill, mp, fp_mp,
14555                                     nce->nice_fp_mp != NULL ?
14556                                     MBLKL(nce->nice_fp_mp) : 0);
14557                 return (EWOULDBLOCK);
14558             }
14559             BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCOutTransmits);
14560             UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCOutOctets,
14561                      pkt_len);
14562             putnext(wq, mp);
14563         }
14565
14565         /*
14566          * The rest of this function implements Neighbor Unreachability
14567          * detection. Determine if the ncec is eligible for NUD.
14568          */
14569         if (ncec->ncec_flags & NCE_F_NONUD)
14570             return (0);
14572
14572         ASSERT(ncec->ncec_state != ND_INCOMPLETE);
14574
14574         /*
14575          * Check for upper layer advice
14576          */
14577         if (ixaflags & IXAF_REACH_CONF) {
14578             timeout_id_t tid;
14580
14580             /*
14581              * It should be o.k. to check the state without

```

```

14582             * a lock here, at most we lose an advice.
14583             */
14584             ncec->ncec_last = TICK_TO_MSEC(now);
14585             if (ncec->ncec_state != ND_REACHABLE) {
14586                 mutex_enter(&ncec->ncec_lock);
14587                 ncec->ncec_state = ND_REACHABLE;
14588                 tid = ncec->ncec_timeout_id;
14589                 ncec->ncec_timeout_id = 0;
14590                 mutex_exit(&ncec->ncec_lock);
14591                 (void) untimeout(tid);
14592                 if (ip_debug > 2) {
14593                     /* ip1dbg */
14594                     pr_addr_dbg("ip_xmit: state"
14595                                " for %s changed to"
14596                                " REACHABLE\n", AF_INET6,
14597                                &ncec->ncec_addr);
14598                 }
14599             }
14600             return (0);
14601         }
14603
14603         delta = TICK_TO_MSEC(now) - ncec->ncec_last;
14604         ip1dbg(("ip_xmit: delta = %" PRId64
14605              " ill_reachable_time = %d\n", delta,
14606              ill->ill_reachable_time));
14607         if (delta > (uint64_t)ill->ill_reachable_time) {
14608             mutex_enter(&ncec->ncec_lock);
14609             switch (ncec->ncec_state) {
14610                 case ND_REACHABLE:
14611                     ASSERT((ncec->ncec_flags & NCE_F_NONUD) == 0);
14612                     /* FALLTHROUGH */
14613                 case ND_STALE:
14614                     /*
14615                      * ND_REACHABLE is identical to
14616                      * ND_STALE in this specific case. If
14617                      * reachable time has expired for this
14618                      * neighbor (delta is greater than
14619                      * reachable time), conceptually, the
14620                      * neighbor cache is no longer in
14621                      * REACHABLE state, but already in
14622                      * STALE state. So the correct
14623                      * transition here is to ND_DELAY.
14624                      */
14625                     ncec->ncec_state = ND_DELAY;
14626                     mutex_exit(&ncec->ncec_lock);
14627                     nce_restart_timer(ncec,
14628                                     ipst->ips_delay_first_probe_time);
14629                     if (ip_debug > 3) {
14630                         /* ip2dbg */
14631                         pr_addr_dbg("ip_xmit: state"
14632                                    " for %s changed to"
14633                                    " DELAY\n", AF_INET6,
14634                                    &ncec->ncec_addr);
14635                     }
14636                     break;
14637                 case ND_DELAY:
14638                 case ND_PROBE:
14639                     mutex_exit(&ncec->ncec_lock);
14640                     /* Timers have already started */
14641                     break;
14642                 case ND_UNREACHABLE:
14643                     /*
14644                      * nce_timer has detected that this ncec
14645                      * is unreachable and initiated deleting
14646                      * this ncec.
14647                      * This is a harmless race where we found the

```

```

14648         * ncec before it was deleted and have
14649         * just sent out a packet using this
14650         * unreachable ncec.
14651         */
14652         mutex_exit(&ncec->ncec_lock);
14653         break;
14654     default:
14655         ASSERT(0);
14656         mutex_exit(&ncec->ncec_lock);
14657     }
14658     return (0);
14659
14661 case ND_INCOMPLETE:
14662     /*
14663     * the state could have changed since we didn't hold the lock.
14664     * Re-verify state under lock.
14665     */
14666     is_probe = ipmp_packet_is_probe(mp, nce->ncc_ill);
14667     mutex_enter(&ncec->ncec_lock);
14668     if (NCE_ISREACHABLE(ncec)) {
14669         mutex_exit(&ncec->ncec_lock);
14670         goto sendit;
14671     }
14672     /* queue the packet */
14673     nce_queue_mp(ncec, mp, is_probe);
14674     mutex_exit(&ncec->ncec_lock);
14675     DTRACE_PROBE2(ip_xmit_incomplete,
14676     (ncec_t *), ncec, (mblk_t *), mp);
14677     return (0);
14679 case ND_INITIAL:
14680     /*
14681     * State could have changed since we didn't hold the lock, so
14682     * re-verify state.
14683     */
14684     is_probe = ipmp_packet_is_probe(mp, nce->ncc_ill);
14685     mutex_enter(&ncec->ncec_lock);
14686     if (NCE_ISREACHABLE(ncec)) {
14687         mutex_exit(&ncec->ncec_lock);
14688         goto sendit;
14689     }
14690     nce_queue_mp(ncec, mp, is_probe);
14691     if (ncec->ncec_state == ND_INITIAL) {
14692         ncec->ncec_state = ND_INCOMPLETE;
14693         mutex_exit(&ncec->ncec_lock);
14694         /*
14695         * figure out the source we want to use
14696         * and resolve it.
14697         */
14698         ip_ndp_resolve(ncec);
14699     } else {
14700         mutex_exit(&ncec->ncec_lock);
14701     }
14702     return (0);
14704 case ND_UNREACHABLE:
14705     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
14706     ip_drop_output("ipIfStatsOutDiscards - ND_UNREACHABLE",
14707     mp, ill);
14708     freemsg(mp);
14709     return (0);
14711 default:
14712     ASSERT(0);
14713     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);

```

```

14714         ip_drop_output("ipIfStatsOutDiscards - ND_other",
14715         mp, ill);
14716         freemsg(mp);
14717         return (ENETUNREACH);
14718     }
14719 }
14721 /*
14722 * Return B_TRUE if the buffers differ in length or content.
14723 * This is used for comparing extension header buffers.
14724 * Note that an extension header would be declared different
14725 * even if all that changed was the next header value in that header i.e.
14726 * what really changed is the next extension header.
14727 */
14728 boolean_t
14729 ip_cmpbuf(const void *abuf, uint_t alen, boolean_t b_valid, const void *bbuf,
14730 uint_t blen)
14731 {
14732     if (!b_valid)
14733         blen = 0;
14735     if (alen != blen)
14736         return (B_TRUE);
14737     if (alen == 0)
14738         return (B_FALSE); /* Both zero length */
14739     return (bcmp(abuf, bbuf, alen));
14740 }
14742 /*
14743 * Preallocate memory for ip_savebuf(). Returns B_TRUE if ok.
14744 * Return B_FALSE if memory allocation fails - don't change any state!
14745 */
14746 boolean_t
14747 ip_allocbuf(void **dstp, uint_t *dstlenp, boolean_t src_valid,
14748 const void *src, uint_t srclen)
14749 {
14750     void *dst;
14752     if (!src_valid)
14753         srclen = 0;
14755     ASSERT(*dstlenp == 0);
14756     if (src != NULL && srclen != 0) {
14757         dst = mi_alloc(srclen, BPRI_MED);
14758         if (dst == NULL)
14759             return (B_FALSE);
14760     } else {
14761         dst = NULL;
14762     }
14763     if (*dstp != NULL)
14764         mi_free(*dstp);
14765     *dstp = dst;
14766     *dstlenp = dst == NULL ? 0 : srclen;
14767     return (B_TRUE);
14768 }
14770 /*
14771 * Replace what is in *dst, *dstlen with the source.
14772 * Assumes ip_allocbuf has already been called.
14773 */
14774 void
14775 ip_savebuf(void **dstp, uint_t *dstlenp, boolean_t src_valid,
14776 const void *src, uint_t srclen)
14777 {
14778     if (!src_valid)
14779         srclen = 0;

```

```

14781     ASSERT(*dstlenp == srclen);
14782     if (src != NULL && srclen != 0)
14783         bcopy(src, *dstp, srclen);
14784 }

14786 /*
14787  * Free the storage pointed to by the members of an ip_pkt_t.
14788  */
14789 void
14790 ip_pkt_free(ip_pkt_t *ipp)
14791 {
14792     uint_t fields = ipp->ipp_fields;

14794     if (fields & IPPF_HOPOPTS) {
14795         kmem_free(ipp->ipp_hopopts, ipp->ipp_hopoptslen);
14796         ipp->ipp_hopopts = NULL;
14797         ipp->ipp_hopoptslen = 0;
14798     }
14799     if (fields & IPPF_RTHDRDSTOPTS) {
14800         kmem_free(ipp->ipp_rthdrdstopts, ipp->ipp_rthdrdstoptslen);
14801         ipp->ipp_rthdrdstopts = NULL;
14802         ipp->ipp_rthdrdstoptslen = 0;
14803     }
14804     if (fields & IPPF_DSTOPTS) {
14805         kmem_free(ipp->ipp_dstopts, ipp->ipp_dstoptslen);
14806         ipp->ipp_dstopts = NULL;
14807         ipp->ipp_dstoptslen = 0;
14808     }
14809     if (fields & IPPF_RTHDR) {
14810         kmem_free(ipp->ipp_rthdr, ipp->ipp_rthdrlen);
14811         ipp->ipp_rthdr = NULL;
14812         ipp->ipp_rthdrlen = 0;
14813     }
14814     if (fields & IPPF_IPV4_OPTIONS) {
14815         kmem_free(ipp->ipp_ipv4_options, ipp->ipp_ipv4_options_len);
14816         ipp->ipp_ipv4_options = NULL;
14817         ipp->ipp_ipv4_options_len = 0;
14818     }
14819     if (fields & IPPF_LABEL_V4) {
14820         kmem_free(ipp->ipp_label_v4, ipp->ipp_label_len_v4);
14821         ipp->ipp_label_v4 = NULL;
14822         ipp->ipp_label_len_v4 = 0;
14823     }
14824     if (fields & IPPF_LABEL_V6) {
14825         kmem_free(ipp->ipp_label_v6, ipp->ipp_label_len_v6);
14826         ipp->ipp_label_v6 = NULL;
14827         ipp->ipp_label_len_v6 = 0;
14828     }
14829     ipp->ipp_fields &= ~(IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
14830         IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6);
14831 }

14833 /*
14834  * Copy from src to dst and allocate as needed.
14835  * Returns zero or ENOMEM.
14836  *
14837  * The caller must initialize dst to zero.
14838  */
14839 int
14840 ip_pkt_copy(ip_pkt_t *src, ip_pkt_t *dst, int kmflag)
14841 {
14842     uint_t fields = src->ipp_fields;

14844     /* Start with fields that don't require memory allocation */
14845     dst->ipp_fields = fields &

```

```

14846     ~(IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
14847         IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6);

14849     dst->ipp_addr = src->ipp_addr;
14850     dst->ipp_unicast_hops = src->ipp_unicast_hops;
14851     dst->ipp_hoplimit = src->ipp_hoplimit;
14852     dst->ipp_tclass = src->ipp_tclass;
14853     dst->ipp_type_of_service = src->ipp_type_of_service;

14855     if (!(fields & (IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
14856         IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6)))
14857         return (0);

14859     if (fields & IPPF_HOPOPTS) {
14860         dst->ipp_hopopts = kmem_alloc(src->ipp_hopoptslen, kmflag);
14861         if (dst->ipp_hopopts == NULL) {
14862             ip_pkt_free(dst);
14863             return (ENOMEM);
14864         }
14865         dst->ipp_fields |= IPPF_HOPOPTS;
14866         bcopy(src->ipp_hopopts, dst->ipp_hopopts,
14867             src->ipp_hopoptslen);
14868         dst->ipp_hopoptslen = src->ipp_hopoptslen;
14869     }
14870     if (fields & IPPF_RTHDRDSTOPTS) {
14871         dst->ipp_rthdrdstopts = kmem_alloc(src->ipp_rthdrdstoptslen,
14872             kmflag);
14873         if (dst->ipp_rthdrdstopts == NULL) {
14874             ip_pkt_free(dst);
14875             return (ENOMEM);
14876         }
14877         dst->ipp_fields |= IPPF_RTHDRDSTOPTS;
14878         bcopy(src->ipp_rthdrdstopts, dst->ipp_rthdrdstopts,
14879             src->ipp_rthdrdstoptslen);
14880         dst->ipp_rthdrdstoptslen = src->ipp_rthdrdstoptslen;
14881     }
14882     if (fields & IPPF_DSTOPTS) {
14883         dst->ipp_dstopts = kmem_alloc(src->ipp_dstoptslen, kmflag);
14884         if (dst->ipp_dstopts == NULL) {
14885             ip_pkt_free(dst);
14886             return (ENOMEM);
14887         }
14888         dst->ipp_fields |= IPPF_DSTOPTS;
14889         bcopy(src->ipp_dstopts, dst->ipp_dstopts,
14890             src->ipp_dstoptslen);
14891         dst->ipp_dstoptslen = src->ipp_dstoptslen;
14892     }
14893     if (fields & IPPF_RTHDR) {
14894         dst->ipp_rthdr = kmem_alloc(src->ipp_rthdrlen, kmflag);
14895         if (dst->ipp_rthdr == NULL) {
14896             ip_pkt_free(dst);
14897             return (ENOMEM);
14898         }
14899         dst->ipp_fields |= IPPF_RTHDR;
14900         bcopy(src->ipp_rthdr, dst->ipp_rthdr,
14901             src->ipp_rthdrlen);
14902         dst->ipp_rthdrlen = src->ipp_rthdrlen;
14903     }
14904     if (fields & IPPF_IPV4_OPTIONS) {
14905         dst->ipp_ipv4_options = kmem_alloc(src->ipp_ipv4_options_len,
14906             kmflag);
14907         if (dst->ipp_ipv4_options == NULL) {
14908             ip_pkt_free(dst);
14909             return (ENOMEM);
14910         }
14911         dst->ipp_fields |= IPPF_IPV4_OPTIONS;

```



```

15044         break;
15045     }
15046 }
15047 }

15049 /*
15050  * Returns NULL if no routing header
15051  */
15052 in6_addr_t *
15053 ip_pkt_source_route_v6(const ip_pkt_t *ipp)
15054 {
15055     in6_addr_t     *nexthop = NULL;
15056     ip6_rthdr0_t   *rthdr;

15058     if (!(ipp->ipp_fields & IPPF_RTHDR))
15059         return (NULL);

15061     rthdr = (ip6_rthdr0_t *)ipp->ipp_rthdr;
15062     if (rthdr->ip6r0_segleft == 0)
15063         return (NULL);

15065     nexthop = (in6_addr_t *)((char *)rthdr + sizeof (*rthdr));
15066     return (nexthop);
15067 }

15069 zoneid_t
15070 ip_get_zoneid_v4(ipaddr_t addr, mblk_t *mp, ip_recv_attr_t *ira,
15071     zoneid_t lookup_zoneid)
15072 {
15073     ip_stack_t     *ipst = ira->ira_ill->ill_ipst;
15074     ire_t          *ire;
15075     int            ire_flags = MATCH_IRE_TYPE;
15076     zoneid_t      zoneid = ALL_ZONES;

15078     if (is_system_labeled() && !tsol_can_accept_raw(mp, ira, B_FALSE))
15079         return (ALL_ZONES);

15081     if (lookup_zoneid != ALL_ZONES)
15082         ire_flags |= MATCH_IRE_ZONEONLY;
15083     ire = ire_fhtable_lookup_v4(addr, NULL, NULL, IRE_LOCAL | IRE_LOOPBACK,
15084     NULL, lookup_zoneid, NULL, ire_flags, 0, ipst, NULL);
15085     if (ire != NULL) {
15086         zoneid = IP_REAL_ZONEID(ire->ire_zoneid, ipst);
15087         ire_refrele(ire);
15088     }
15089     return (zoneid);
15090 }

15092 zoneid_t
15093 ip_get_zoneid_v6(in6_addr_t *addr, mblk_t *mp, const ill_t *ill,
15094     ip_recv_attr_t *ira, zoneid_t lookup_zoneid)
15095 {
15096     ip_stack_t     *ipst = ira->ira_ill->ill_ipst;
15097     ire_t          *ire;
15098     int            ire_flags = MATCH_IRE_TYPE;
15099     zoneid_t      zoneid = ALL_ZONES;

15101     if (is_system_labeled() && !tsol_can_accept_raw(mp, ira, B_FALSE))
15102         return (ALL_ZONES);

15104     if (IN6_IS_ADDR_LINKLOCAL(addr))
15105         ire_flags |= MATCH_IRE_ILL;

15107     if (lookup_zoneid != ALL_ZONES)
15108         ire_flags |= MATCH_IRE_ZONEONLY;
15109     ire = ire_fhtable_lookup_v6(addr, NULL, NULL, IRE_LOCAL | IRE_LOOPBACK,

```

```

15110     ill, lookup_zoneid, NULL, ire_flags, 0, ipst, NULL);
15111     if (ire != NULL) {
15112         zoneid = IP_REAL_ZONEID(ire->ire_zoneid, ipst);
15113         ire_refrele(ire);
15114     }
15115     return (zoneid);
15116 }

15118 /*
15119  * IP obserability hook support functions.
15120  */
15121 static void
15122 ipobs_init(ip_stack_t *ipst)
15123 {
15124     netid_t id;

15126     id = net_getnetidbynetstackid(ipst->ips_netstack->netstack_stackid);

15128     ipst->ips_ip4_observe_pr = net_protocol_lookup(id, NHF_INET);
15129     VERIFY(ipst->ips_ip4_observe_pr != NULL);

15131     ipst->ips_ip6_observe_pr = net_protocol_lookup(id, NHF_INET6);
15132     VERIFY(ipst->ips_ip6_observe_pr != NULL);
15133 }

15135 static void
15136 ipobs_fini(ip_stack_t *ipst)
15137 {
15139     VERIFY(net_protocol_release(ipst->ips_ip4_observe_pr) == 0);
15140     VERIFY(net_protocol_release(ipst->ips_ip6_observe_pr) == 0);
15141 }

15143 /*
15144  * hook_pkt_observe_t is composed in network byte order so that the
15145  * entire mblk_t chain handed into hook_run can be used as-is.
15146  * The caveat is that use of the fields, such as the zone fields,
15147  * requires conversion into host byte order first.
15148  */
15149 void
15150 ipobs_hook(mblk_t *mp, int htype, zoneid_t zsrc, zoneid_t zdst,
15151     const ill_t *ill, ip_stack_t *ipst)
15152 {
15153     hook_pkt_observe_t *hdr;
15154     uint64_t grifindex;
15155     mblk_t *imp;

15157     imp = allocb(sizeof (*hdr), BPRI_HI);
15158     if (imp == NULL)
15159         return;

15161     hdr = (hook_pkt_observe_t *)imp->b_rptr;
15162     /*
15163      * b_wptr is set to make the apparent size of the data in the mblk_t
15164      * to exclude the pointers at the end of hook_pkt_observer_t.
15165      */
15166     imp->b_wptr = imp->b_rptr + sizeof (dl_ipnetinfo_t);
15167     imp->b_cont = mp;

15169     ASSERT(DB_TYPE(mp) == M_DATA);

15171     if (IS_UNDER_IPMP(ill))
15172         grifindex = ipmp_ill_get_ipmp_ifindex(ill);
15173     else
15174         grifindex = 0;

```

```

15176     hdr->hpo_version = 1;
15177     hdr->hpo_hatype = htons(htype);
15178     hdr->hpo_pktlen = htonl((ulong_t)msgdsize(mp));
15179     hdr->hpo_ifindex = htonl(ill->ill_phyint->phyint_ifindex);
15180     hdr->hpo_grifindex = htonl(grifindex);
15181     hdr->hpo_zsrc = htonl(zsrc);
15182     hdr->hpo_zdst = htonl(zdst);
15183     hdr->hpo_pkt = imp;
15184     hdr->hpo_ctx = ipst->ips_netstack;

15186     if (ill->ill_isv6) {
15187         hdr->hpo_family = AF_INET6;
15188         (void) hook_run(ipst->ips_ipv6_net_data->netd_hooks,
15189             ipst->ips_ipv6observing, (hook_data_t)hdr);
15190     } else {
15191         hdr->hpo_family = AF_INET;
15192         (void) hook_run(ipst->ips_ipv4_net_data->netd_hooks,
15193             ipst->ips_ipv4observing, (hook_data_t)hdr);
15194     }

15196     imp->b_cont = NULL;
15197     freemsg(imp);
15198 }

15200 /*
15201  * Utility routine that checks if 'v4srcp' is a valid address on underlying
15202  * interface 'ill'. If 'ipifp' is non-NULL, it's set to a held ipif
15203  * associated with 'v4srcp' on success. NOTE: if this is not called from
15204  * inside the IPSQ (ill_g_lock is not held), 'ill' may be removed from the
15205  * group during or after this lookup.
15206  */
15207 boolean_t
15208 ipif_lookup_testaddr_v4(ill_t *ill, const in_addr_t *v4srcp, ipif_t **ipifp)
15209 {
15210     ipif_t *ipif;

15212     ipif = ipif_lookup_addr_exact(*v4srcp, ill, ill->ill_ipst);
15213     if (ipif != NULL) {
15214         if (ipifp != NULL)
15215             *ipifp = ipif;
15216         else
15217             ipif_refrele(ipif);
15218         return (B_TRUE);
15219     }

15221     ipldbg(("ipif_lookup_testaddr_v4: cannot find ipif for src %x\n",
15222         *v4srcp));
15223     return (B_FALSE);
15224 }

15226 /*
15227  * Transport protocol call back function for CPU state change.
15228  */
15229 /* ARGSUSED */
15230 static int
15231 ip_tp_cpu_update(cpu_setup_t what, int id, void *arg)
15232 {
15233     processorid_t cpu_seqid;
15234     netstack_handle_t nh;
15235     netstack_t *ns;

15237     ASSERT(MUTEX_HELD(&cpu_lock));

15239     switch (what) {
15240     case CPU_CONFIG:
15241     case CPU_ON:

```

```

15242     case CPU_INIT:
15243     case CPU_CPUPART_IN:
15244         cpu_seqid = cpu[id]->cpu_seqid;
15245         netstack_next_init(&nh);
15246         while ((ns = netstack_next(&nh)) != NULL) {
15247             tcp_stack_cpu_add(ns->netstack_tcp, cpu_seqid);
15248             sctp_stack_cpu_add(ns->netstack_sctp, cpu_seqid);
15249             udp_stack_cpu_add(ns->netstack_udp, cpu_seqid);
15250             netstack_rele(ns);
15251         }
15252         netstack_next_fini(&nh);
15253         break;
15254     case CPU_UNCONFIG:
15255     case CPU_OFF:
15256     case CPU_CPUPART_OUT:
15257         /*
15258          * Nothing to do. We don't remove the per CPU stats from
15259          * the IP stack even when the CPU goes offline.
15260          */
15261         break;
15262     default:
15263         break;
15264     }
15265     return (0);
15266 }

```

new/usr/src/uts/common/inet/ip/ip_if.c

1

```
*****
533855 Wed Jun 13 12:04:52 2012
new/usr/src/uts/common/inet/ip/ip_if.c
%B
*****
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 */
21 /*
22 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 1990 Mentat Inc.
24 */

26 /*
27 * This file contains the interface control functions for IP.
28 */

30 #include <sys/types.h>
31 #include <sys/stream.h>
32 #include <sys/dlpi.h>
33 #include <sys/stropts.h>
34 #include <sys/strsun.h>
35 #include <sys/sysmacros.h>
36 #include <sys/strsubr.h>
37 #include <sys/strlog.h>
38 #include <sys/ddi.h>
39 #include <sys/sunddi.h>
40 #include <sys/cmn_err.h>
41 #include <sys/kstat.h>
42 #include <sys/debug.h>
43 #include <sys/zone.h>
44 #include <sys/sunldi.h>
45 #include <sys/file.h>
46 #include <sys/bitmap.h>
47 #include <sys/cpuvar.h>
48 #include <sys/time.h>
49 #include <sys/ctype.h>
50 #include <sys/kmem.h>
51 #include <sys/system.h>
52 #include <sys/param.h>
53 #include <sys/socket.h>
54 #include <sys/isa_defs.h>
55 #include <net/if.h>
56 #include <net/if_arp.h>
57 #include <net/if_types.h>
58 #include <net/if_dl.h>
59 #include <net/route.h>
60 #include <sys/sockio.h>
61 #include <netinet/in.h>
```

new/usr/src/uts/common/inet/ip/ip_if.c

2

```
62 #include <netinet/ip6.h>
63 #include <netinet/icmp6.h>
64 #include <netinet/igmp_var.h>
65 #include <sys/policy.h>
66 #include <sys/ethernet.h>
67 #include <sys/callb.h>
68 #include <sys/md5.h>

70 #include <inet/common.h> /* for various inet/mi.h and inet/nd.h needs */
71 #include <inet/mi.h>
72 #include <inet/nd.h>
73 #include <inet/tunables.h>
74 #include <inet/arp.h>
75 #include <inet/ip_arp.h>
76 #include <inet/mib2.h>
77 #include <inet/ip.h>
78 #include <inet/ip6.h>
79 #include <inet/ip6_asp.h>
80 #include <inet/tcp.h>
81 #include <inet/ip_multi.h>
82 #include <inet/ip_ire.h>
83 #include <inet/ip_ftable.h>
84 #include <inet/ip_rts.h>
85 #include <inet/ip_ndp.h>
86 #include <inet/ip_if.h>
87 #include <inet/ip_impl.h>
88 #include <inet/sctp_ip.h>
89 #include <inet/ip_netinfo.h>
90 #include <inet/ilb_ip.h>

92 #include <netinet/igmp.h>
93 #include <inet/ip_listutils.h>
94 #include <inet/ipclassifier.h>
95 #include <sys/mac_client.h>
96 #include <sys/dld.h>
97 #include <sys/mac_flow.h>

99 #include <sys/systeminfo.h>
100 #include <sys/bootconf.h>

102 #include <sys/tsol/tndb.h>
103 #include <sys/tsol/tnet.h>

105 #include <inet/rawip_impl.h> /* needed for icmp_stack_t */
106 #include <inet/udp_impl.h> /* needed for udp_stack_t */
107 #include <inet/dccp/dccp_stack.h> /* needed for dccp_stack_t */
108 #endif /* ! codereview */

110 /* The character which tells where the ill_name ends */
111 #define IPIF_SEPARATOR_CHAR ':'

113 /* IP ioctl function table entry */
114 typedef struct ipft_s {
115     int ipft_cmd;
116     pfi_t ipft_pfi;
117     int ipft_min_size;
118     int ipft_flags;
119 } ipft_t;
120 #define IPFT_F_NO_REPLY 0x1 /* IP ioctl does not expect any reply */
121 #define IPFT_F_SELF_REPLY 0x2 /* ioctl callee does the ioctl reply */

123 static int nd_ill_forward_get(queue_t *, mblk_t *, caddr_t, cred_t *);
124 static int nd_ill_forward_set(queue_t *q, mblk_t *mp,
125     char *value, caddr_t cp, cred_t *ioc_cr);

127 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_mbcst_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_usersrc(ill_t *);
216 static int ill_relink_usersrc_ills(ill_t *, ill_t *, uint_t);
217 static void ill_disband_usersrc_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_INETADDRESSSTYPE_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_INETADDRESSSTYPE_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 phynt 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 mouted */
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 ioctl's
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_down1 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 *mpl;

622             mp = mpp[0];
623             mpp[0] = mp->b_next;
624             for (mpl = mp; mpl != NULL; mpl = mpl->b_cont) {
625                 mpl->b_next = NULL;
626                 mpl->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         }
664     }
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 }
675
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;
689
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     } else if (sap_length > 0) {
701         bcopy(&sap_addr, (char *)dst + phys_length, sizeof (sap_addr));
702     } else {
703         bcopy(&sap_addr, dst, sizeof (sap_addr));
704         if (phys_src == NULL)
705             bzero((char *)dst + sap_length, phys_length);
706         else
707             bcopy(phys_src, (char *)dst + sap_length, phys_length);
708     }
709 }
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 */
724
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 }
742
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;
756
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);
768
769     /*
770      * M_IOCTLDATA from ioctls, M_ERROR/M_HANGUP/M_PROTO/M_PCPROTO from the
771      * driver.
772      */
773     ASSERT((DB_TYPE(add_mp) == M_IOCTLDATA) || (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));
776
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  * ipx_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 ipx_pending_mp_get(ipx_t *ipx, conn_t **connpp)
813 {
814     mblk_t *curr = NULL;
815     ipxop_t *ipx = ipx->ipxop;

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 rehold 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 ipx_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_ipx->ipxop;

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 *, "ipx_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_IOCTLDATA) {
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;
1091
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;
1107
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;
1141
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);
1159
1160     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next)
1161         (void) ipif_down(ipif, NULL, NULL);
1162
1163     ill_down(ill);
1164
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);
1170
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);
1174
1175     ipsq_current_start(ill->ill_phyint->phyint_ipsq, ill->ill_ipif, 0);
1176
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 = mkioch(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     ipldbg(("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 renege)
1383 {
1384     ASSERT(IAM_WRITER_ILL(ill));

1386     if (ill->ill_dlpi_capab_state != IDCS_OK)
1387         return;

1389     ill->ill_dlpi_capab_state = renege ? 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         ipldbg(("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         ipldbg(("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         ipldbg(("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         ipldbg(("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         ipldbg(("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         ipldbg(("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         ipldbg(("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 *rprr;

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         rprr = nmp->b_rptr;
1840         /* initialize dl_capability_req_t */
1841         oc = (dl_capability_req_t *) rprr;
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     ipldbg(("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         ipldbg(("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     ipldbg(("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         ipldbg(("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         ipldbg(("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 reneq;

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 isn't 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         reneq = ill->ill_dlpi_capab_state == IDCS_RENEG;
2291         ill_capability_dld_disable(ill);
2292         ill->ill_dlpi_capab_state = IDCS_UNKNOWN;
2293         if (reneq)
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 }
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 = gethrstime_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_rcv_attr_t iras;
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;
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;
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;
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;
2516
2517     mp = send_icmp_head;
2518     send_icmp_head = send_icmp_head->b_next;
2519     mp->b_next = NULL;
2520
2521     dst = ((ipha_t *)mp->b_rptr)->ipha_dst;
2522
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  * A non-dying ILL will use the return value to decide whether to
2538  * restart the frag timer, and for how long.
2539  */
2540 return (next_timeout);
2541 }
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;
2555
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)) {
2566
2567         ill->ill_frag_free_num_pkts++;
2568
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);
2574
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;
2580
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;
2599
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;
2614
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 }
2625
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     ipldb("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->ncc_common;
2753             mutex_enter(&ncec->ncc_lock);
2754             if (enable)
2755                 ncec->ncc_flags |= NCE_F_ISROUTER;
2756             else
2757                 ncec->ncc_flags &= ~NCE_F_ISROUTER;
2758             mutex_exit(&ncec->ncc_lock);
2759             nce_refrele(nce);
2760         }
2761     }
2762 }

2764 /*
2765  * Initializes the context structure and returns the first ill in the list
2766  * currently start_list and end_list can have values:
2767  * MAX_G_HEADS          Traverse both IPV4 and IPV6 lists.

```



```

2768 * IP_V4_G_HEAD      Traverse IPv4 list only.
2769 * IP_V6_G_HEAD      Traverse IPv6 list only.
2770 */

2772 /*
2773 * We don't check for CONDEMNED ill_s here. Caller must do that if
2774 * necessary under the ill lock.
2775 */
2776 ill_t *
2777 ill_first(int start_list, int end_list, ill_walk_context_t *ctx,
2778           ip_stack_t *ipst)
2779 {
2780     ill_if_t *ifp;
2781     ill_t *ill;
2782     avl_tree_t *avl_tree;

2784     ASSERT(RW_LOCK_HELD(&ipst->ips_ill_g_lock));
2785     ASSERT(end_list <= MAX_G_HEADS && start_list >= 0);

2787     /*
2788      * setup the lists to search
2789      */
2790     if (end_list != MAX_G_HEADS) {
2791         ctx->ctx_current_list = start_list;
2792         ctx->ctx_last_list = end_list;
2793     } else {
2794         ctx->ctx_last_list = MAX_G_HEADS - 1;
2795         ctx->ctx_current_list = 0;
2796     }

2798     while (ctx->ctx_current_list <= ctx->ctx_last_list) {
2799         ifp = IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst);
2800         if (ifp != (ill_if_t *)
2801             &IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst)) {
2802             avl_tree = &ifp->illif_avl_by_ppa;
2803             ill = avl_first(avl_tree);
2804             /*
2805              * ill is guaranteed to be non NULL or ifp should have
2806              * not existed.
2807              */
2808             ASSERT(ill != NULL);
2809             return (ill);
2810         }
2811         ctx->ctx_current_list++;
2812     }

2814     return (NULL);
2815 }

2817 /*
2818 * returns the next ill in the list. ill_first() must have been called
2819 * before calling ill_next() or bad things will happen.
2820 */

2822 /*
2823 * We don't check for CONDEMNED ill_s here. Caller must do that if
2824 * necessary under the ill lock.
2825 */
2826 ill_t *
2827 ill_next(ill_walk_context_t *ctx, ill_t *lastill)
2828 {
2829     ill_if_t *ifp;
2830     ill_t *ill;
2831     ip_stack_t *ipst = lastill->ill_ipst;

2833     ASSERT(lastill->ill_ifptr != (ill_if_t *)

```

```

2834         &IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst));
2835     if ((ill = avl_walk(&lastill->ill_ifptr->illif_avl_by_ppa, lastill,
2836                       AVL_AFTER)) != NULL) {
2837         return (ill);
2838     }

2840     /* goto next ill_ifp in the list. */
2841     ifp = lastill->ill_ifptr->illif_next;

2843     /* make sure not at end of circular list */
2844     while (ifp ==
2845            (ill_if_t *)&IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst)) {
2846         if (++ctx->ctx_current_list > ctx->ctx_last_list)
2847             return (NULL);
2848         ifp = IP_VX_ILL_G_LIST(ctx->ctx_current_list, ipst);
2849     }

2851     return (avl_first(&ifp->illif_avl_by_ppa));
2852 }

2854 /*
2855 * Check interface name for correct format: [a-zA-Z]+[a-zA-Z0-9_]*[0-9]+
2856 * The final number (PPA) must not have any leading zeros. Upon success, a
2857 * pointer to the start of the PPA is returned; otherwise NULL is returned.
2858 */
2859 static char *
2860 ill_get_ppa_ptr(char *name)
2861 {
2862     int namelen = strlen(name);
2863     int end_ndx = namelen - 1;
2864     int ppa_ndx, i;

2866     /*
2867      * Check that the first character is [a-zA-Z], and that the last
2868      * character is [0-9].
2869      */
2870     if (namelen == 0 || !isalpha(name[0]) || !isdigit(name[end_ndx]))
2871         return (NULL);

2873     /*
2874      * Set 'ppa_ndx' to the PPA start, and check for leading zeroes.
2875      */
2876     for (ppa_ndx = end_ndx; ppa_ndx > 0; ppa_ndx--)
2877         if (!isdigit(name[ppa_ndx - 1]))
2878             break;

2880     if (name[ppa_ndx] == '0' && ppa_ndx < end_ndx)
2881         return (NULL);

2883     /*
2884      * Check that the intermediate characters are [a-z0-9_].
2885      */
2886     for (i = 1; i < ppa_ndx; i++) {
2887         if (!isalpha(name[i]) && !isdigit(name[i]) &&
2888             name[i] != '.' && name[i] != '_') {
2889             return (NULL);
2890         }
2891     }

2893     return (name + ppa_ndx);
2894 }

2896 /*
2897 * use avl tree to locate the ill.
2898 */
2899 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;
2908
2909     /*
2910      * get ppa ptr
2911      */
2912     if (isv6)
2913         list = IP_V6_G_HEAD;
2914     else
2915         list = IP_V4_G_HEAD;
2916
2917     if ((ppa_ptr = ill_get_ppa_ptr(name)) == NULL) {
2918         return (NULL);
2919     }
2920
2921     len = ppa_ptr - name + 1;
2922
2923     ppa = stoi(&ppa_ptr);
2924
2925     ifp = IP_VX_ILL_G_LIST(list, ipst);
2926
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     }
2940
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     }
2947
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 }
2960
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;
2969
2970     ASSERT(ppa_ptr != NULL && ill_ptr != NULL);
2971
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 }
2984
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);
2993
2994     avl_destroy(&interface->illif_avl_by_ppa);
2995     if (interface->illif_ppa_arena != NULL)
2996         vmem_destroy(interface->illif_ppa_arena);
2997
2998     remque(interface);
2999
3000     mi_free(interface);
3001 }
3002
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;
3011
3012     if (ill == NULL)
3013         return;
3014     ipst = ill->ill_ipst;
3015     rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
3016
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);
3026
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     /*
3160     * No arena is in use and not enough (>ill_no_arena) interfaces have
3161     * been plumbed to create one. Do a linear search to get a unused ppa.
3162     */
3163     if (ill->ill_ppa == UINT_MAX) {
3164         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 chosen 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     ddir = (dl_info_req_t *)info_mp->b_rptr;
3446     info_mp->b_wptr = (uchar_t *)&ddir[1];
3447     ddir->dl_primitive = DL_INFO_REQ;

3449     ill->ill_dlpi_pending = DL_PRIM_INVALID;

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     }
3584 }

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 ill's. 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_INVALID;

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     cmm_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 primarily 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_sell_t *sell;
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         ipldbg(("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 when 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 receiving 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 underlying 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     sell = (dl_qos_cl_sell_t *)mi_offset_param(mp, dlia->dl_qos_offset,
4315         dlia->dl_qos_length);
4316     if ((sell != NULL) && (sell->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);

4355     if (CLASSD(addr))
4356         return (B_FALSE);

4358     return (B_TRUE);
4359 }

4361 #define V6_IPIF_LINKLOCAL(p) \
4362     IN6_IS_ADDR_LINKLOCAL(&(p)->ipif_v6lcl_addr)

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 account) 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     }

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     }

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     }

4403     /* old_ipif is a normal interface, so no need to use the new one. */
4404     return (B_FALSE);
4405 }

4407 /*
4408  * Find a multicast-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;

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);

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             }

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 *dlinp;
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         dlinp = (dl_notify_ind_t *)mp->b_rptr;
4997         ASSERT(dlinp->dl_primitive == DL_NOTIFY_IND);
4999         switch (dlinp->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_IOCTLDATA:
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 /*
5386  * The netmask can be verified to be contiguous with 32 shifts and or
5387  * operations. Take the contiguous mask (in host byte order) and compute
5388  * mask | mask << 1 | mask << 2 | ... | mask << 31
5389  * the result will be the same as the 'mask' for contiguous mask.
5390  */
5391 static boolean_t
5392 ip_contiguous_mask(uint32_t mask)
5393 {
5394     uint32_t m = mask;
5395     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 rtas_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     ipldbg(("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);

5447     /*
5448      * Get the ipif, if any, corresponding to the gw_addr
5449      * If -ifp was specified we restrict ourselves to the ill, otherwise
5450      * we match on the gateway and destination to handle unnumbered pt-pt
5451      * interfaces.
5452      */
5453     if (ill != NULL)
5454         ipif = ipif_lookup_addr(gw_addr, ill, ALL_ZONES, ipst);
5455     else
5456         ipif = ipif_lookup_interface(gw_addr, dst_addr, ipst);
5457     if (ipif != NULL) {
5458         if (IS_VNI(ipif->ipif_ill)) {
5459             ipif_refrele(ipif);
5460             return (EINVAL);
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_fhtable_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     ipldbg(("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_fstable_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     ipldbg(("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_fhtable_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_fhtable_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_fhtable_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 *mpl;
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     ipldbg(("ip_siocaddrt:"));
6124     /* Existence of mpl verified in ip_wput_nondata */
6125     mpl = mp->b_cont->b_cont;
6126     rt = (struct rtentry *)mpl->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 *mpl;
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     ipldbg(("ip_siocdelrt:"));
6176     /* Existence of mpl verified in ip_wput_nondata */
6177     mpl = mp->b_cont->b_cont;
6178     rt = (struct rtentry *)mpl->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_mthead != NULL);
6232             ipx->ipx_mptail->b_next = mp;
6233         } else {
6234             ASSERT(ipx->ipx_mthead == NULL);
6235             ipx->ipx_mthead = 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_mthead != NULL);
6243             ipsq->ipsq_xopq_mptail->b_next = mp;
6244         } else {
6245             ASSERT(ipsq->ipsq_xopq_mthead == NULL);
6246             ipsq->ipsq_xopq_mthead = 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_mthead), 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_mthead.
6291  * Note that if the phynt 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 phynt in the group, including the IPMP meta-interface phynt.
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         phynt_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_mthead) != NULL) {
6322             ipx->ipx_mthead = mp->b_next;
6323             if (ipx->ipx_mthead == 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_ioctl_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_PROBEL(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_mthead == 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 = getpstack(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_mthead) != NULL) {
6468         xopipsq->ipsq_xopq_mthead = mp->b_next;
6469         if (xopipsq->ipsq_xopq_mthead == 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_INVALID ||
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_INVALID)
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 don't 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 ill's 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_mthead == 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_enqueue(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_mthead. 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_mthead.
6727  */

```

```

6728 * If a thread does not want to reenter the ipsq when it is already writer,
6729 * it must make sure that the specified reentry point to be called later
6730 * when the ipsq is empty, nor any code path starting from the specified reentry
6731 * point must never ever try to enter the ipsq again. Otherwise it can lead
6732 * to an infinite loop. The reentry point ip_rput_dlpi_writer is an example.
6733 * When the thread that is currently exclusive finishes, it (ipsq_exit)
6734 * dequeues the requests waiting to become exclusive in ipx_mthead and calls
6735 * the reentry point. When the list at ipx_mthead becomes empty ipsq_exit
6736 * proceeds to dequeue the next ioctl in ipsq_xopq_mthead and start the next
6737 * ioctl if the current ioctl has completed. If the current ioctl is still
6738 * in progress it simply returns. The current ioctl could be waiting for
6739 * a response from another module (the driver or could be waiting for
6740 * the ipif/ill/ire refcnts to drop to zero. In such a case the ipx_pending_mp
6741 * and ipx_pending_ipif are set. ipx_current_ipif is set throughout the
6742 * execution of the ioctl and ipsq_exit does not start the next ioctl unless
6743 * ipx_current_ipif is NULL which happens only once the ioctl is complete and
6744 * all associated DLPI operations have completed.
6745 */

6747 /*
6748 * Try to enter the IPSQ corresponding to 'ipif' or 'ill' exclusively ('ipif'
6749 * and 'ill' cannot both be specified). Returns a pointer to the entered IPSQ
6750 * on success, or NULL on failure. The caller ensures ipif/ill is valid by
6751 * reholding it as necessary. If the IPSQ cannot be entered and 'func' is
6752 * non-NULL, then 'func' will be called back with 'q' and 'mp' once the IPSQ
6753 * can be entered. If 'func' is NULL, then 'q' and 'mp' are ignored.
6754 */
6755 ipsq_t *
6756 ipsq_try_enter(ipif_t *ipif, ill_t *ill, queue_t *q, mblk_t *mp,
6757               ipsq_func_t func, int type, boolean_t reentry_ok)
6758 {
6759     ip_stack_t    *ipst;
6760     ipsq_t        *ipsq;

6762     /* Only 1 of ipif or ill can be specified */
6763     ASSERT((ipif != NULL) ^ (ill != NULL));

6765     if (ipif != NULL)
6766         ill = ipif->ipif_ill;
6767     ipst = ill->ill_ipst;

6769     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
6770     ipsq = ipsq_try_enter_internal(ill, q, mp, func, type, reentry_ok);
6771     rw_exit(&ipst->ips_ill_g_lock);

6773     return (ipsq);
6774 }

6776 /*
6777 * Try to enter the IPSQ corresponding to 'ill' as writer. The caller ensures
6778 * ill is valid by reholding it if necessary; we will refrele. If the IPSQ
6779 * cannot be entered, the mp is queued for completion.
6780 */
6781 void
6782 qwriter_ip(ill_t *ill, queue_t *q, mblk_t *mp, ipsq_func_t func, int type,
6783           boolean_t reentry_ok)
6784 {
6785     ipsq_t *ipsq;

6787     ipsq = ipsq_try_enter(NULL, ill, q, mp, func, type, reentry_ok);

6789     /*
6790     * Drop the caller's rehold on the ill. This is safe since we either
6791     * entered the IPSQ (and thus are exclusive), or failed to enter the
6792     * IPSQ, in which case we return without accessing ill anymore. This
6793     * is needed because func needs to see the correct refcount.

```

```

6794     * e.g. removeif can work only then.
6795     */
6796     ill_refrele(ill);
6797     if (ipsq != NULL) {
6798         (*func)(ipsq, q, mp, NULL);
6799         ipsq_exit(ipsq);
6800     }
6801 }

6803 /*
6804 * Exit the specified IPSQ. If this is the final exit on it then drain it
6805 * prior to exiting. Caller must be writer on the specified IPSQ.
6806 */
6807 void
6808 ipsq_exit(ipsq_t *ipsq)
6809 {
6810     mblk_t *mp;
6811     ipsq_t *mp_ipsq;
6812     queue_t *q;
6813     phyint_t *phyi;
6814     ipsq_func_t func;

6816     ASSERT(IAM_WRITER_IPSQ(ipsq));

6818     ASSERT(ipsq->ipsq_xop->ipx_reentry_cnt >= 1);
6819     if (ipsq->ipsq_xop->ipx_reentry_cnt != 1) {
6820         ipsq->ipsq_xop->ipx_reentry_cnt--;
6821         return;
6822     }

6824     for (;;) {
6825         phyi = ipsq->ipsq_phyint;
6826         mp = ipsq_dq(ipsq);
6827         mp_ipsq = (mp == NULL) ? NULL : (ipsq_t *)mp->b_next;

6829         /*
6830         * If we've changed to a new IPSQ, and the phyint associated
6831         * with the old one has gone away, free the old IPSQ. Note
6832         * that this cannot happen while the IPSQ is in a group.
6833         */
6834         if (mp_ipsq != ipsq && phyi == NULL) {
6835             ASSERT(ipsq->ipsq_next == ipsq);
6836             ASSERT(ipsq->ipsq_xop == &ipsq->ipsq_ownxop);
6837             ipsq_delete(ipsq);
6838         }

6840         if (mp == NULL)
6841             break;

6843         q = mp->b_queue;
6844         func = (ipsq_func_t)mp->b_prev;
6845         ipsq = mp_ipsq;
6846         mp->b_next = mp->b_prev = NULL;
6847         mp->b_queue = NULL;

6849         /*
6850         * If 'q' is a conn queue, it is valid, since we did a
6851         * a rehold on the conn at the start of the ioctl.
6852         * If 'q' is an ill queue, it is valid, since close of an
6853         * ill will clean up its IPSQ.
6854         */
6855         (*func)(ipsq, q, mp, NULL);
6856     }
6857 }

6859 /*

```

```

6860 * Used to start any igmp or mld timers that could not be started
6861 * while holding ill_mcast_lock. The timers can't be started while holding
6862 * the lock, since mld/igmp_start_timers may need to call untimeout()
6863 * which can't be done while holding the lock which the timeout handler
6864 * acquires. Otherwise
6865 * there could be a deadlock since the timeout handlers
6866 * mld_timeout_handler_per_ill/igmp_timeout_handler_per_ill also acquire
6867 * ill_mcast_lock.
6868 */
6869 void
6870 ill_mcast_timer_start(ip_stack_t *ipst)
6871 {
6872     int         next;

6874     mutex_enter(&ipst->ips_igmp_timer_lock);
6875     next = ipst->ips_igmp_deferred_next;
6876     ipst->ips_igmp_deferred_next = INFINITY;
6877     mutex_exit(&ipst->ips_igmp_timer_lock);

6879     if (next != INFINITY)
6880         igmp_start_timers(next, ipst);

6882     mutex_enter(&ipst->ips_mld_timer_lock);
6883     next = ipst->ips_mld_deferred_next;
6884     ipst->ips_mld_deferred_next = INFINITY;
6885     mutex_exit(&ipst->ips_mld_timer_lock);

6887     if (next != INFINITY)
6888         mld_start_timers(next, ipst);
6889 }

6891 /*
6892 * Start the current exclusive operation on 'ipsq'; associate it with 'ipif'
6893 * and 'ioccmd'.
6894 */
6895 void
6896 ipsq_current_start(ipsq_t *ipsq, ipif_t *ipif, int ioccmd)
6897 {
6898     ill_t *ill = ipif->ipif_ill;
6899     ipxop_t *ipx = ipsq->ipsq_xop;

6901     ASSERT(IAM_WRITER_IPSQ(ipsq));
6902     ASSERT(ipx->ipx_current_ipif == NULL);
6903     ASSERT(ipx->ipx_current_ioctl == 0);

6905     ipx->ipx_current_done = B_FALSE;
6906     ipx->ipx_current_ioctl = ioccmd;
6907     mutex_enter(&ipx->ipx_lock);
6908     ipx->ipx_current_ipif = ipif;
6909     mutex_exit(&ipx->ipx_lock);

6911     /*
6912     * Set IPIF_CHANGING on one or more ipifs associated with the
6913     * current exclusive operation. IPIF_CHANGING prevents any new
6914     * references to the ipif (so that the references will eventually
6915     * drop to zero) and also prevents any "get" operations (e.g.,
6916     * SIOCGLIFFLAGS) from being able to access the ipif until the
6917     * operation has completed and the ipif is again in a stable state.
6918     *
6919     * For ioctls, IPIF_CHANGING is set on the ipif associated with the
6920     * ioctl. For internal operations (where ioccmd is zero), all ipifs
6921     * on the ill are marked with IPIF_CHANGING since it's unclear which
6922     * ipifs will be affected.
6923     *
6924     * Note that SIOCLIFREMOVEIF is a special case as it sets
6925     * IPIF_CONDEMNED internally after identifying the right ipif to

```

```

6926     * operate on.
6927     */
6928     switch (ioccmd) {
6929     case SIOCLIFREMOVEIF:
6930         break;
6931     case 0:
6932         mutex_enter(&ill->ill_lock);
6933         ipif = ipif->ipif_ill->ill_ipif;
6934         for (; ipif != NULL; ipif = ipif->ipif_next)
6935             ipif->ipif_state_flags |= IPIF_CHANGING;
6936         mutex_exit(&ill->ill_lock);
6937         break;
6938     default:
6939         mutex_enter(&ill->ill_lock);
6940         ipif->ipif_state_flags |= IPIF_CHANGING;
6941         mutex_exit(&ill->ill_lock);
6942     }
6943 }

6945 /*
6946 * Finish the current exclusive operation on 'ipsq'. Usually, this will allow
6947 * the next exclusive operation to begin once we ipsq_exit(). However, if
6948 * pending DLPI operations remain, then we will wait for the queue to drain
6949 * before allowing the next exclusive operation to begin. This ensures that
6950 * DLPI operations from one exclusive operation are never improperly processed
6951 * as part of a subsequent exclusive operation.
6952 */
6953 void
6954 ipsq_current_finish(ipsq_t *ipsq)
6955 {
6956     ipxop_t *ipx = ipsq->ipsq_xop;
6957     t_uscalar_t dlpi_pending = DL_PRIM_INVAL;
6958     ipif_t *ipif = ipx->ipx_current_ipif;

6960     ASSERT(IAM_WRITER_IPSQ(ipsq));

6962     /*
6963     * For SIOCLIFREMOVEIF, the ipif has been already been blown away
6964     * (but in that case, IPIF_CHANGING will already be clear and no
6965     * pending DLPI messages can remain).
6966     */
6967     if (ipx->ipx_current_ioctl != SIOCLIFREMOVEIF) {
6968         ill_t *ill = ipif->ipif_ill;

6970         mutex_enter(&ill->ill_lock);
6971         dlpi_pending = ill->ill_dlpi_pending;
6972         if (ipx->ipx_current_ioctl == 0) {
6973             ipif = ill->ill_ipif;
6974             for (; ipif != NULL; ipif = ipif->ipif_next)
6975                 ipif->ipif_state_flags &= ~IPIF_CHANGING;
6976         } else {
6977             ipif->ipif_state_flags &= ~IPIF_CHANGING;
6978         }
6979         mutex_exit(&ill->ill_lock);
6980     }

6982     ASSERT(!ipx->ipx_current_done);
6983     ipx->ipx_current_done = B_TRUE;
6984     ipx->ipx_current_ioctl = 0;
6985     if (dlpi_pending == DL_PRIM_INVAL) {
6986         mutex_enter(&ipx->ipx_lock);
6987         ipx->ipx_current_ipif = NULL;
6988         mutex_exit(&ipx->ipx_lock);
6989     }
6990 }

```



```

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_mthead
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_mthead; 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_mthead = 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 rehold
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 *pip,
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 *mpl;
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     mpl = mp->b_cont->b_cont;
7081     if (pip->ipi_cmd_type == IF_CMD) {
7082         /* This a old style SIOC[GS]IF* command */
7083         ifr = (struct ifreq *)mpl->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[LIFNAMSIZ - 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(pip->ipi_cmd_type == LIF_CMD);
7097         lifr = (struct lifreq *)mpl->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 (pip->ipi_cmd == SIOC[S]LIFNAME) {
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_rehold(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         ipldbg(("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 *pip, 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     ipldbg(("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 *pip, void *ifreq)
7297 {
7298     struct lifnum *lifn;
7299     mblk_t *mpl;
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     mpl = mp->b_cont->b_cont;

7307     lifn = (struct lifnum *)mpl->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     ipldbg(("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 *pip, void *ifreq)
7327 {
7328     STRUCT_HANDLE(ifconf, ifc);
7329     mblk_t *mpl;
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 ifc_len;
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     ipldbg(("ip_sioctl_get_ifconf"));
7343     /* Existence verified in ip_wput_nondata */
7344     mpl = 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     STRUCT_SET_HANDLE(ifc, iocp->ioc_flag, NULL);
7368     if ((mpl->b_wptr - mpl->b_rptr) == STRUCT_SIZE(ifc)) {
7369         int numifs = 0;
7370         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 *)mpl->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 have 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      ifcflen = STRUCT_FGET(ifc, ifc_len);
7397      ifc_bufsize = numifs * sizeof (struct ifreq);
7398      if (ifc_bufsize > ifcflen) {
7399          if (iocp->ioc_cmd == O_SIOCGIFCONF) {
7400              /* old behaviour */
7401              return (EINVAL);
7402          } else {
7403              ifc_bufsize = ifcflen;
7404          }
7405      }

7407      mpl = mi_copyout_alloc(q, mp,
7408                          STRUCT_FGETP(ifc, ifc_buf, ifc_bufsize, B_FALSE);
7409      if (mpl == NULL)
7410          return (ENOMEM);

7412      mpl->b_wptr = mpl->b_rptr + ifc_bufsize;
7413  }
7414  bzero(mpl->b_rptr, mpl->b_wptr - mpl->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 *)mpl->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] > mpl->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      mpl->b_wptr = (uchar_t *)ifr;

7456      if (STRUCT_BUF(ifc) != NULL) {
7457          STRUCT_FSET(ifc, ifc_len,
7458                    (int)((uchar_t *)ifr - mpl->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 address
7466   */
7467  /* ARGSUSED */
7468  int
7469  ip_ioctl_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 *mpl;
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_nodata */
7494      mpl = 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 *)mpl->b_rptr);

7503      if (MBLKL(mpl) != 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          ipldbg(("ip_ioctl_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     mpl = mi_copyout_alloc(q, mp,
7529         STRUCT_FGETP(lifs, lifs_buf), lifs_bufsize, B_FALSE);
7530     if (mpl == NULL) {
7531         ipif_refrele(ipif);
7532         return (ENOMEM);
7533     }
7535     mpl->b_wptr = mpl->b_rptr + lifs_bufsize;
7536     bzero(mpl->b_rptr, lifs_bufsize);
7538     lifr = (struct lifreq *)mpl->b_rptr;
7540     ill = ill_head = ipif->ipif_ill;
7541     orig_ipif = ipif;
7543     /* ill_g_usersrc_lock protects ill_usersrc_grp_next */
7544     rw_enter(&ipst->ips_ill_g_usersrc_lock, RW_READER);
7545     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
7547     ill = ill->ill_usersrc_grp_next; /* start from next ill */
7548     for (; (ill != NULL) && (ill != ill_head);
7549         ill = ill->ill_usersrc_grp_next) {
7551         if ((uchar_t *)&lifr[1] > mpl->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_usersrc_lock);
7575     ipif_refrele(orig_ipif);
7576     mpl->b_wptr = (uchar_t *)lifr;
7577     STRUCT_FSET(lifs, lifs_len, (int)((uchar_t *)lifr - mpl->b_rptr));
7579     return (0);
7580 }
7582 /* ARGSUSED */
7583 int
7584 ip_ioctl_get_lifconf(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7585     mblk_t *mp, ip_ioctl_cmd_t *pip, void *ifreq)

```

```

7586 {
7587     mblk_t *mpl;
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 lifcflen;
7601     zoneid_t zoneid;
7602     STRUCT_HANDLE(lifconf, lifc);
7603     ip_stack_t *ipst = CONNQ_TO_IPST(q);
7605     ipldbq(("ip_ioctl_get_lifconf"));
7607     ASSERT(q->q_next == NULL);
7609     zoneid = Q_TO_CONN(q)->conn_zoneid;
7611     /* Existence verified in ip_wput_nodata */
7612     mpl = 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 ((mpl->b_wptr - mpl->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 *)mpl->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     mpl = mi_copyout_alloc(q, mp,
7684         STRUCT_FGETP(lifc, lifc_buf), lifc_bufsize, B_FALSE);
7685     if (mpl == NULL)
7686         return (ENOMEM);

7688     mpl->b_wptr = mpl->b_rptr + lifc_bufsize;
7689     bzero(mpl->b_rptr, mpl->b_wptr - mpl->b_rptr);

7691     lifr = (struct lifreq *)mpl->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] > mpl->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     mpl->b_wptr = (uchar_t *)lifr;
7761     if (STRUCT_BUF(lifc) != NULL) {
7762         STRUCT_FSET(lifc, lifc_len,
7763             (int)((uchar_t *)lifr - mpl->b_rptr));
7764     }
7765     return (0);
7766 }

7768 static void
7769 ip_ioctl_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 isip4;
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;
7921
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);
7929
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;
7947
7948             /* move on to next dst addr */
7949             continue;
7950         }
7951         dir->dir_dreachable = 1;
7952
7953         dst_ill = ire_nexthop_ill(ire);
7954         if (dst_ill == NULL) {
7955             ire_refrele(ire);
7956             continue;
7957         }
7958
7959         /* With ipmp we most likely look at the ipmp ill here */
7960         dir->dir_dmactype = dst_ill->ill_mactype;
7961
7962         if (isipv4) {
7963             ipaddr_t v4saddr;
7964
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         }
7980         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 }
7990
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_ioctl_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 *mpl;
8013     zoneid_t zoneid;
8014     ip_stack_t *ipst;
8015
8016     ipldbg(("ip_ioctl_tmyaddr"));
8017
8018     ASSERT(q->q_next == NULL); /* this ioctl not allowed if ip is module */
8019     zoneid = Q_TO_CONN(q)->conn_zoneid;
8020     ipst = CONN_TO_IPST(q);
8021
8022     /* Existence verified in ip_wput_nondata */
8023     mpl = mp->b_cont->b_cont;
8024     sia = (struct sioc_addrreq *)mpl->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;
8029
8030         if (IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
8031             ipaddr_t v4_addr;
8032
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;
8040
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_fhtable_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_ioctl_tonlink(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
8086     ip_ioctl_cmd_t *pip, void *dummy_ifreq)
8087 {
8088     struct sioc_addrreq *sia;
8089     sin_t *sin;
8090     mblk_t *mpl;
8091     ire_t *ire = NULL;
8092     zoneid_t zoneid;
8093     ip_stack_t *ipst;

8095     ipldb("ip_ioctl_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     mpl = mp->b_cont->b_cont;
8103     sia = (struct sioc_addrreq *)mpl->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_fhtable_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_fhtable_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_fhtable_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 maintains a list of site prefixes.
8163  */
8164 /* ARGSUSED */
8165 int
8166 ip_ioctl_tmysite(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
8167     ip_ioctl_cmd_t *pip, void *ifreq)
8168 {
8169     return (ENXIO);
8170 }

8172 /* ARP IOCTLS. */
8173 /* ARGSUSED */
8174 int
8175 ip_ioctl_arp(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
8176     ip_ioctl_cmd_t *pip, 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 != SIOC?DARP) &&
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->ncc_common;

8298 switch (iocp->ioc_cmd) {
8299 case SIOC?DARP:
8300 case SIOC?DXARP: {
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(nccec);
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_ioctl_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(nccec)) {
8344         nce_refrele(nccec);
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 (ncec != NULL) {
8362         nce_refrele(nccec);
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(nccec, ND_UNCHANGED, lladdr);

```

```

8378         mutex_exit(&ncec->ncec_lock);
8379         err = 0;
8380     }
8381     if (nce != NULL) {
8382         nce_refrele(nccec);
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(nccec);
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 rehold 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 *pip,
8416     cmd_info_t *ci)
8417 {
8418     mblk_t *mpl;
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     mpl = mp->b_cont->b_cont;

8441     if (pip->ipi_cmd_type == XARP_CMD) {
8442         ASSERT(MBLKL(mpl) >= sizeof (struct xarpreq));
8443         xar = (struct xarpreq *)mpl->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(mpl) >= sizeof (struct arpreq));
8454     ar = (struct arpreq *)mpl->b_rprtr;
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 SIOC[LIF]GROUPNAME 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_ioctl_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 SIOC[LIF]GROUPNAME 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 SIOC[LIF]GROUPNAME 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);

8566     default:
8567         break;
8568     }
8569     return (0);
8570 }

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

```

```

8576 * its clones use the persistent link, while pppd(1M) and perhaps many
8577 * other daemons may use non-persistent link. When combined with some
8578 * ill_t states, linking and unlinking lower streams may be used as
8579 * indicators of dynamic re-plumbing events [see PSARC/1999/348].
8580 */
8581 /* ARGSUSED */
8582 void
8583 ip_ioctl_plink(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
8584 {
8585     mblk_t      *mpl;
8586     struct linkblk *li;
8587     int         ioccmd = ((struct iocblk *)mp->b_rptr)->ioc_cmd;
8588     int         err = 0;

8590     ASSERT(ioccmd == I_PLINK || ioccmd == I_PUNLINK ||
8591           ioccmd == I_LINK || ioccmd == I_UNLINK);

8593     mpl = mp->b_cont; /* This is the linkblk info */
8594     li = (struct linkblk *)mpl->b_rptr;

8596     err = ip_ioctl_plink_ipmod(ipsq, q, mp, ioccmd, li);
8597     if (err == EINPROGRESS)
8598         return;
8599     if (err == 0)
8600         miocack(q, mp, 0, 0);
8601     else
8602         miocnak(q, mp, 0, err);

8604     /* Conn was refheld in ip_ioctl_copyin_setup */
8605     if (CONN_Q(q)) {
8606         CONN_DEC_IOCTLREF(Q_TO_CONN(q));
8607         CONN_OPER_PENDING_DONE(Q_TO_CONN(q));
8608     }
8609 }

8611 /*
8612 * Process I_{P}LINK and I_{P}UNLINK requests named by 'ioccmd' and pointed to
8613 * by 'mp' and 'li' for the IP module stream (if li->q_bot is in fact an IP
8614 * module stream).
8615 * Returns zero on success, EINPROGRESS if the operation is still pending, or
8616 * an error code on failure.
8617 */
8618 static int
8619 ip_ioctl_plink_ipmod(ipsq_t *ipsq, queue_t *q, mblk_t *mp, int ioccmd,
8620                     struct linkblk *li)
8621 {
8622     int         err = 0;
8623     ill_t      *ill;
8624     queue_t    *ipwq, *dwq;
8625     const char *name;
8626     struct qinit *qinfo;
8627     boolean_t  islink = (ioccmd == I_PLINK || ioccmd == I_LINK);
8628     boolean_t  entered_ipsq = B_FALSE;
8629     boolean_t  is_ip = B_FALSE;
8630     arl_t      *arl;

8632     /*
8633     * Walk the lower stream to verify it's the IP module stream.
8634     * The IP module is identified by its name, wput function,
8635     * and non-NULL q_next. STREAMS ensures that the lower stream
8636     * (li->l_qbot) will not vanish until this ioctl completes.
8637     */
8638     for (ipwq = li->l_qbot; ipwq != NULL; ipwq = ipwq->q_next) {
8639         qinfo = ipwq->q_qinfo;
8640         name = qinfo->q_minfo->mi_idname;
8641         if (name != NULL && strcmp(name, ip_mod_info.mi_idname) == 0 &&

```

```

8642         qinfo->q_i_putp != (pfi_t)ip_lwput && ipwq->q_next != NULL) {
8643             is_ip = B_TRUE;
8644             break;
8645         }
8646         if (name != NULL && strcmp(name, arp_mod_info.mi_idname) == 0 &&
8647             qinfo->q_i_putp != (pfi_t)ip_lwput && ipwq->q_next != NULL) {
8648             break;
8649         }
8650     }

8652     /*
8653     * If this isn't an IP module stream, bail.
8654     */
8655     if (ipwq == NULL)
8656         return (0);

8658     if (!is_ip) {
8659         arl = (arl_t *)ipwq->q_ptr;
8660         ill = arl_to_ill(arl);
8661         if (ill == NULL)
8662             return (0);
8663     } else {
8664         ill = ipwq->q_ptr;
8665     }
8666     ASSERT(ill != NULL);

8668     if (ipsq == NULL) {
8669         ipsq = ipsq_try_enter(NULL, ill, q, mp, ip_ioctl_plink,
8670                               NEW_OP, B_FALSE);
8671         if (ipsq == NULL) {
8672             if (!is_ip)
8673                 ill_refrele(ill);
8674             return (EINPROGRESS);
8675         }
8676         entered_ipsq = B_TRUE;
8677     }
8678     ASSERT(IAM_WRITER_ILL(ill));
8679     mutex_enter(&ill->ill_lock);
8680     if (!is_ip) {
8681         if (islink && ill->ill_muxid == 0) {
8682             /*
8683             * Plumbing has to be done with IP plumbed first, arp
8684             * second, but here we have arp being plumbed first.
8685             */
8686             mutex_exit(&ill->ill_lock);
8687             if (entered_ipsq)
8688                 ipsq_exit(ipsq);
8689             ill_refrele(ill);
8690             return (EINVAL);
8691         }
8692     }
8693     mutex_exit(&ill->ill_lock);
8694     if (!is_ip) {
8695         arl->arl_muxid = islink ? li->l_index : 0;
8696         ill_refrele(ill);
8697         goto done;
8698     }

8700     if (IS_IPMP(ill) && (err = ip_ioctl_plink_ipmp(ill, ioccmd)) != 0)
8701         goto done;

8703     /*
8704     * As part of I_{P}LINKing, stash the number of downstream modules and
8705     * the read queue of the module immediately below IP in the ill.
8706     * These are used during the capability negotiation below.
8707     */

```

```

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 *mpl = 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 (mpl == NULL || iocp->ioc_count == 0 ||
8800         iocp->ioc_count == TRANSPARENT) {
8801         return (B_FALSE);
8802     }
8803     if (msgdsz(mpl) < iocp->ioc_count) {
8804         if (!pullupmsg(mpl, iocp->ioc_count))
8805             return (B_FALSE);
8806     }

8808     pioc = (mod_ioc_prop_t *)mpl->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     }
8843
8844     return (B_TRUE);
8845 }
8846
8847 /*
8848  * process the SIOC{SET|GET}PROP ioctl's
8849  */
8850 /* ARGSUSED */
8851 static void
8852 ip_ioctl_getsetprop(queue_t *q, mblk_t *mp)
8853 {
8854     struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
8855     mblk_t *mpl = 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;
8870
8871     ASSERT(q->q_next == NULL);
8872     ASSERT(CONN_Q(q));
8873
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 *)mpl->b_rptr;
8881
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     }
8918
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     }
8929
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     }
8943
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 }
8953
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 *mpl = 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;
8970
8971     ASSERT(CONN_Q(q));

```

```

8972     ipst = CONNQ_TO_IPST(q);
8974     if (iocp->ioc_count == 0 || mpl == NULL) {
8975         miocnak(q, mp, 0, EINVAL);
8976         return;
8977     }
8979     mpl->b_datap->db_lim[-1] = '\0';          /* Force null termination */
8980     pval = buf = pname = (char *)mpl->b_rptr;
8981     bufsize = MBLKL(mpl);
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 *)mpl->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_ioctl_copyin_resume(ipsq_t *dummy_ipsq, queue_t *q, mblk_t *mp,
9043      void *dummy_arg)
9044  {
9045      ip_ioctl_copyin_setup(q, mp);
9046  }
9048  /*
9049  * ip_ioctl_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_IOCADATA message.
9055  */
9056  void
9057  ip_ioctl_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 *pip;
9062      cred_t *cr;
9063      ip_stack_t *ipst;
9065      if (CONNQ(q))
9066          ipst = CONNQ_TO_IPST(q);
9067      else
9068          ipst = ILLQ_TO_IPST(q);
9070      pip = ip_ioctl_lookup(iocp->ioc_cmd);
9071      if (pip == 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_ioctl_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     }

9109     /* Make sure we have ioctl data to process. */
9110     if (mp->b_cont == NULL && !(ipip->ipi_flags & IPI_NULL_BCONT))
9111         goto nak;

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 cfree()) 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;

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     }

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     }

9143     switch (iocp->ioc_cmd) {
9144     case O_SIOCGIFCONF:
9145     case SIOCGIFCONF:
9146         /*
9147         * This IOCTL is hilarious. See comments in
9148         * ip_ioctl_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;

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;

9164     case SIOCGLIFSRCONF:
9165         copyin_size = SIZEOF_STRUCT(lifsrcf, iocp->ioc_flag);
9166         mi_copyin(q, mp, NULL, copyin_size);
9167         return;

9169     case SIOCGIP6ADDRPOLICY:

```

```

9170         ip_ioctl_ip6addrpolicy(q, mp);
9171         ip6_asp_table_refrele(ipst);
9172         return;

9174     case SIOCSIP6ADDRPOLICY:
9175         ip_ioctl_ip6addrpolicy(q, mp);
9176         return;

9178     case SIOCGDSTINFO:
9179         ip_ioctl_dstinfo(q, mp);
9180         ip6_asp_table_refrele(ipst);
9181         return;

9183     case ND_SET:
9184     case ND_GET:
9185         ip_process_legacy_nddprop(q, mp);
9186         return;

9188     case SIOCSETPROP:
9189     case SIOCGETPROP:
9190         ip_ioctl_getsetprop(q, mp);
9191         return;

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_ioctl_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_ioctl_plink(NULL, q, mp, NULL);
9211         return;

9213     case IP_IOCTL:
9214         ip_wput_ioctl(q, mp);
9215         return;

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;

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_ioctl_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_ioctl_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_ioctl_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 *mpl;
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     ipldbq("ip_ioctl_addif\n");
9339     /* Existence of mpl has been checked in ip_wput_nondata */
9340     mpl = 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 *)mpl->b_rptr;
9347     lifr->lifr_name[LIFNAMSIZ - 1] = '\0';
9348     name = lifr->lifr_name;
9349     ASSERT(CONNQ(q));
9350     connp = Q_TO_CONNQ(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);
9395
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 }
9410
9411 ipsq = ipsq_try_enter(NULL, ill, q, mp, ip_process_ioctl, NEW_OP,
9412     B_TRUE);
9413
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);
9419
9420 if (ipsq == NULL)
9421     return (EINPROGRESS);
9422
9423 /* We are now exclusive on the IPSQ */
9424 ASSERT(IAM_WRITER_ILL(ill));
9425
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     }

```

```

9434     }
9435 }
9436
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 }
9446
9447 /* Return created name with ioctl */
9448 (void) sprintf(lifr->lifr_name, "%s%d", ill->ill_name,
9449     IPIF_SEPARATOR_CHAR, ipif->ipif_id);
9450 ipldb("created %s\n", lifr->lifr_name);
9451
9452 /* Set address */
9453 sin = (sin_t *)&lifr->lifr_addr;
9454 if (sin->sin_family != AF_UNSPEC) {
9455     err = ip_ioctl_addr(ipif, sin, q, mp,
9456         &ip_ndx_ioctl_table[SIOCLIFADDR_NDX], lifr);
9457 }
9458
9459 done:
9460     ipsq_exit(ipsq);
9461     return (err);
9462 }
9463
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_ioctl_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;
9479
9480     ipst = CONNQ_TO_IPST(q);
9481
9482     ASSERT(q->q_next == NULL);
9483     ipldb("ip_ioctl_remove_if(%s:%u %p)\n",
9484         ill->ill_name, ipif->ipif_id, (void *)ipif);
9485     ASSERT(IAM_WRITER_IPIF(ipif));
9486
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_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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_ioctl_prefix_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9692                       ip_ioctl_cmd_t *ipip, void *ifreq)
9693 {
9694     ipldbg(("ip_ioctl_prefix_restart(%s:%u %p)\n",
9695           ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
9696     return (ip_ioctl_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_ioctl_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     ipldbg(("ip_ioctl_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) &&

```

```

9764     IN6_IS_ADDR_UNSPECIFIED(&v6addr) &&
9765     (!(ipif->ipif_flags & IPIF_NOLOCAL) ||
9766     IN6_IS_ADDR_UNSPECIFIED(&ipif->ipif_v6subnet))) {
9767         return (EADDRNOTAVAIL);
9768     }

9770     if (!ip_local_addr_ok_v6(&v6addr, &ipif->ipif_v6net_mask))
9771         return (EADDRNOTAVAIL);
9772 } else {
9773     ipaddr_t addr;

9775     if (sin->sin_family != AF_INET)
9776         return (EAFNOSUPPORT);

9778     addr = sin->sin_addr.s_addr;

9780     /* Allow INADDR_ANY as the local address. */
9781     if (addr != INADDR_ANY &&
9782         !ip_addr_ok_v4(addr, ipif->ipif_net_mask))
9783         return (EADDRNOTAVAIL);

9785     IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
9786 }
9787 /*
9788  * verify that the address being configured is permitted by the
9789  * ill_allowed_ips[] for the interface.
9790  */
9791 if (ill->ill_allowed_ips_cnt > 0) {
9792     for (i = 0; i < ill->ill_allowed_ips_cnt; i++) {
9793         if (IN6_ARE_ADDR_EQUAL(&ill->ill_allowed_ips[i],
9794             &v6addr))
9795             break;
9796     }
9797     if (i == ill->ill_allowed_ips_cnt) {
9798         pr_addr_dbg("I allowed addr %s\n", AF_INET6, &v6addr);
9799         return (EPERM);
9800     }
9801 }
9802 /*
9803  * Even if there is no change we redo things just to rerun
9804  * ipif_set_default.
9805  */
9806 if (ipif->ipif_flags & IPIF_UP) {
9807     /*
9808      * Setting a new local address, make sure
9809      * we have net and subnet broadcast addresses for
9810      * the old address if we need them.
9811      */
9812     /*
9813      * If the interface is already marked up,
9814      * we call ipif_down which will take care
9815      * of ditching any IREs that have been set
9816      * up based on the old interface address.
9817      */
9818     err = ipif_logical_down(ipif, q, mp);
9819     if (err == EINPROGRESS)
9820         return (err);
9821     (void) ipif_down_tail(ipif);
9822     need_up = 1;
9823 }

9825 err = ip_ioctl_addr_tail(ipif, sin, q, mp, need_up);
9826 return (err);
9827 }

9829 int

```

```

9830 ip_ioctl_addr_tail(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9831     boolean_t need_up)
9832 {
9833     in6_addr_t v6addr;
9834     in6_addr_t ov6addr;
9835     ipaddr_t addr;
9836     sin6_t *sin6;
9837     int sinlen;
9838     int err = 0;
9839     ill_t *ill = ipif->ipif_ill;
9840     boolean_t need_dl_down;
9841     boolean_t need_arp_down;
9842     struct iocblk *iocp;

9844     iocp = (mp != NULL) ? (struct iocblk *)mp->b_rptr : NULL;

9846     ipldbg(("ip_ioctl_addr_tail(%s:%u %p)\n",
9847         ill->ill_name, ipif->ipif_id, (void *)ipif));
9848     ASSERT(IAM_WRITER_IPIF(ipif));

9850     /* Must cancel any pending timer before taking the ill_lock */
9851     if (ipif->ipif_recovery_id != 0)
9852         (void) timeout(ipif->ipif_recovery_id);
9853     ipif->ipif_recovery_id = 0;

9855     if (ipif->ipif_isv6) {
9856         sin6 = (sin6_t *)sin;
9857         v6addr = sin6->sin6_addr;
9858         sinlen = sizeof (struct sockaddr_in6);
9859     } else {
9860         addr = sin->sin_addr.s_addr;
9861         IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
9862         sinlen = sizeof (struct sockaddr_in);
9863     }
9864     mutex_enter(&ill->ill_lock);
9865     ov6addr = ipif->ipif_v6lcl_addr;
9866     ipif->ipif_v6lcl_addr = v6addr;
9867     sctp_update_ipif_addr(ipif, ov6addr);
9868     ipif->ipif_addr_ready = 0;

9870     ip_rts_newaddrmsg(RTM_CHGADDR, 0, ipif, RTSQ_DEFAULT);

9872     /*
9873      * If the interface was previously marked as a duplicate, then since
9874      * we've now got a "new" address, it should no longer be considered a
9875      * duplicate -- even if the "new" address is the same as the old one.
9876      * Note that if all ipifs are down, we may have a pending ARP down
9877      * event to handle. This is because we want to recover from duplicates
9878      * and thus delay tearing down ARP until the duplicates have been
9879      * removed or disabled.
9880      */
9881     need_dl_down = need_arp_down = B_FALSE;
9882     if (ipif->ipif_flags & IPIF_DUPLICATE) {
9883         need_arp_down = !need_up;
9884         ipif->ipif_flags &= ~IPIF_DUPLICATE;
9885         if (--ill->ill_ipif_dup_count == 0 && !need_up &&
9886             ill->ill_ipif_up_count == 0 && ill->ill_dl_up) {
9887             need_dl_down = B_TRUE;
9888         }
9889     }

9891     ipif_set_default(ipif);

9893     /*
9894      * If we've just manually set the IPv6 link-local address (0th ipif),
9895      * 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_ioctl_addr_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
9957     ip_ioctl_cmd_t *ipip, void *ifreq)
9958 {
9959     ipldbg(("ip_ioctl_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_ioctl_addr_tail(ipif, sin, q, mp, B_TRUE));
9964 }

9966 /* ARGSUSED */
9967 int
9968 ip_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_dstaddr_tail(ipif, sin, q, mp, need_up);
10062     return (err);
10063 }

10065 static int
10066 ip_ioctl_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     ipldbg(("ip_ioctl_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     ipldbg(("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     ipldbg(("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 SIOCGLIFFLAGS 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     ipldbg(("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_ioctl_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_usersrc_lock here; see the
10415       * synchronization notes in ip.c.
10416       */
10417      if (ill->ill_usersrc_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     }
10426     if (turn_off & PHYI_STANDBY) {
10427         flags &= ~PHYI_INACTIVE;
10428         turn_off |= PHYI_INACTIVE;
10429     }
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);
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     }
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);
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     }
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);
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 }
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     }
10526     /*
10527     * Prevent IPv4 point-to-point interfaces with a 0.0.0.0 destination
10528     * from being brought up.
10529     */
10530     if (lipif->ipif_isv6 &&
10531         ((ipif->ipif_flags & IPIF_POINTOPOINT) &&
10532          ipif->ipif_pp_dst_addr == INADDR_ANY)) {
10533         return (EINVAL);
10534     }
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 bringing 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         ipldbg(("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_ioctl_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_ioctl_flags_tail(ipif, flags, q, mp));
10596 }

10598 static int
10599 ip_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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_ioctl_flags_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10721                       ip_ioctl_cmd_t *pip, void *if_req)
10722 {
10723     uint64_t flags;
10724     struct ifreq *ifreq = if_req;
10725     struct lifreq *lifr = ifreq;
10726     uint64_t turn_on, turn_off;

10728     ipldbg(("ip_ioctl_flags_restart(%s:%u %p)\n",
10729            ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

10731     if (pip->ipi_cmd_type == IF_CMD) {
10732         /* cast to uint16_t prevents unwanted sign extension */
10733         flags = (uint16_t)ifreq->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_ioctl_flags().
10741     */
10742     ip_ioctl_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_ioctl_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_ioctl_get_flags(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10755                   ip_ioctl_cmd_t *pip, 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     ipldbg(("ip_ioctl_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 (pip->ipi_cmd_type == IF_CMD) {
10775         struct ifreq *ifreq = (struct ifreq *)if_req;

10777         /* Get interface flags (low 16 only). */
10778         ifreq->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_ioctl_mtu(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10798             ip_ioctl_cmd_t *pip, void *if_req)
10799 {
10800     int mtu;
10801     int ip_min_mtu;
10802     struct ifreq *ifreq;
10803     struct lifreq *lifr;
10804     ill_t *ill;

10806     ipldbg(("ip_ioctl_mtu(%s:%u %p)\n", ipif->ipif_ill->ill_name,
10807            ipif->ipif_id, (void *)ipif));
10808     if (pip->ipi_cmd_type == IF_CMD) {
10809         ifreq = (struct ifreq *)if_req;
10810         mtu = ifreq->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_ioctl_get_mtu(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10863 ip_ioctl_cmd_t *pip, void *if_req)
10864 {
10865     struct ifreq    *ifr;
10866     struct lifreq   *lifr;

10868     ipldbg(("ip_ioctl_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 (pip->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_ioctl_brdaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10889 ip_ioctl_cmd_t *pip, 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     ipldbg(("ip_ioctl_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_fhtable_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_ioctl_get_brdaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10940 ip_ioctl_cmd_t *pip, void *if_req)
10941 {
10942     ipldbg(("ip_ioctl_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_ioctl_netmask(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10961 ip_ioctl_cmd_t *pip, void *if_req)
10962 {
10963     int err = 0;
10964     in6_addr_t v6mask;

10966     ipldbg(("ip_ioctl_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_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_ioctl_netmask_tail(ipif, sin, q, mp);
11015     return (err);
11016 }

```

```

11018 static int
11019 ip_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_netmask_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11064 ip_ioctl_cmd_t *pip, void *if_req)
11065 {
11066     ipldbg(("ip_ioctl_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_ioctl_netmask_tail(ipif, sin, q, mp));
11070 }

11072 /* Get interface net mask. */
11073 /* ARGSUSED */
11074 int
11075 ip_ioctl_get_netmask(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11076 ip_ioctl_cmd_t *pip, void *if_req)
11077 {
11078     struct lifreq *lifreq = (struct lifreq *)if_req;
11079     struct sockaddr_in6 *sin6 = (sin6_t *)sin;

11081     ipldbg(("ip_ioctl_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_addrflen =
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_addrflen =
11100                  ip_mask_to_plen(ipif->ipif_net_mask);
11101          }
11102      }
11103      return (0);
11104  }

11106 /* ARGSUSED */
11107 int
11108 ip_ioctl_metric(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11109 ip_ioctl_cmd_t *ipip, void *if_req)
11110 {
11111     ipldbg(("ip_ioctl_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_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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     ipldbg(("ip_ioctl_get_muxid(%s:%u %p)\n",
11199 ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
12000     /*
12001      * Get the muxid saved in ill for I_PUNLINK.
12002      */
12003     arp_muxid = arl_get_muxid(ipif->ipif_ill);
12004     if (ipip->ipi_cmd_type == IF_CMD) {
12005         struct ifreq *ifr = (struct ifreq *)if_req;

12007         ifr->ifr_ip_muxid = ipif->ipif_ill->ill_muxid;
12008         ifr->ifr_arp_muxid = arp_muxid;
12009     } else {
12010         struct lifreq *lifr = (struct lifreq *)if_req;

12012         lifr->lifr_ip_muxid = ipif->ipif_ill->ill_muxid;
12013         lifr->lifr_arp_muxid = arp_muxid;
12014     }
12015     return (0);

```



```

11216 }
11218 /*
11219  * Set the subnet prefix. Does not modify the broadcast address.
11220  */
11221 /* ARGSUSED */
11222 int
11223 ip_ioctl_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     ipldbg(("ip_ioctl_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 (IPv6_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_ioctl_subnet_tail(ipif, v6addr, v6mask, q, mp, need_up);
11288     return (err);
11289 }
11291 static int
11292 ip_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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 *lifreq = (struct lifreq *)if_req;
11336     ipldbg(("ip_ioctl_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 = lifreq->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 *lifreq = (struct lifreq *)if_req;
11364     struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *)sin;

11366     ipldbg(("ip_sioctl_get_subnet(%s:%u %p)\n",
11367         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11368     ASSERT(ipip->ipip_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         lifreq->lifreq_addrflen =
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         lifreq->lifreq_addrflen = 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 *ipip, 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 *lifreq = (struct lifreq *)if_req;
11401     int addrlen;

11403     ipldbg(("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 = lifreq->lifreq_addrflen;
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     ipldbg(("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_ioctl_get_token(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11497 ip_ioctl_cmd_t *ipil, void *if_req)
11498 {
11499     ill_t *ill;
11500     sin6_t *sin6 = (sin6_t *)sin;
11501     struct lifreq *lifreq = (struct lifreq *)if_req;

11503     ipldbg(("ip_ioctl_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     lifreq->lifreq_addrflen = 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_ioctl_lnkinfo(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11527 ip_ioctl_cmd_t *ipil, void *if_req)
11528 {
11529     ill_t *ill = ipif->ipif_ill;
11530     int ip_min_mtu;
11531     struct lifreq *lifreq = (struct lifreq *)if_req;
11532     lif_info_req_t *lir;

11534     ipldbg(("ip_ioctl_lnkinfo(%s:%u %p)\n",
11535 ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11536     lir = &lifreq->lifreq_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_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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 *mpl;
11724     cred_t *cr;
11725     int error = 0;
11726     conn_t *connp;

11728     ipldbg(("ip_wput_ioctl"));
11729     iocp = (IOCP)mp->b_rptr;
11730     mpl = mp->b_cont;
11731     if (mpl == NULL) {
11732         iocp->ioc_error = EINVAL;
11733         mp->b_datap->db_type = M_IOCNAK;
11734         iocp->ioc_count = 0;
11735         greply(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(mpl, sizeof (ipllc->ipllc_cmd)))
11750         goto done;

11752     ipllc = (ipllc_t *)mpl->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_ioctl_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         ((mpl->b_wptr - mpl->b_rptr) >= ipft->ipft_min_size ||
11774         pullupmsg(mpl, ipft->ipft_min_size))) {
11775         error = (*ipft->ipft_pfi)(q,
11776             (ipft->ipft_flags & IPFT_F_SELF_REPLY) ? mp : mpl, 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     }
11882
11883     if (virgipif == NULL)
11884         mi_free(sipif);
11885 }
11886
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);
11897
11898     if (ipif_id >= ill->ill_ipst->ips_ip_addrs_per_if)
11899         return (ERANGE);
11900     return (0);
11901 }
11902
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;
11917
11918     ASSERT(ipif->ipif_ill->ill_net_type == IRE_LOOPBACK ||
11919           IAM_WRITER_IPIF(ipif));
11920
11921     ill = ipif->ipif_ill;
11922     ASSERT(ill != NULL);
11923     ipst = ill->ill_ipst;
11924
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 }
11965
11966 ASSERT(tipifp != &(ill->ill_ipif) || id == 0);
11967
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);
11973
11974 return (0);
11975 }
11976
11977 static void
11978 ipif_remove(ipif_t *ipif)
11979 {
11980     ipif_t **ipifp;
11981     ill_t *ill = ipif->ipif_ill;
11982
11983     ASSERT(RW_WRITE_HELD(&ill->ill_ipst->ips_ill_g_lock));
11984
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 }
11995
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 ill's. 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  * Additionally, 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

```

```

12008 * this point. The flags etc. will be set in ip_ll_subnet_defaults when the
12009 * second DL_INFO_ACK comes in from the driver.
12010 */
12011 static ipif_t *
12012 ipif_allocate(ill_t *ill, int id, uint_t ire_type, boolean_t initialize,
12013              boolean_t insert, int *errorp)
12014 {
12015     int err;
12016     ipif_t *ipif;
12017     ip_stack_t *ipst = ill->ill_ipst;
12018
12019     ipldbg(("ipif_allocate(%%s:%%d ill %%p)\n",
12020           ill->ill_name, id, (void *)ill));
12021     ASSERT(ire_type == IRE_LOOPBACK || IAM_WRITER_ILL(ill));
12022
12023     if (errorp != NULL)
12024         *errorp = 0;
12025
12026     if ((ipif = mi_alloc(sizeof(ipif_t), BPRI_MED)) == NULL) {
12027         if (errorp != NULL)
12028             *errorp = ENOMEM;
12029         return (NULL);
12030     }
12031     *ipif = ipif_zero; /* start clean */
12032
12033     ipif->ipif_ill = ill;
12034     ipif->ipif_id = id; /* could be -1 */
12035     /*
12036      * Inherit the zoneid from the ill; for the shared stack instance
12037      * this is always the global zone
12038      */
12039     ipif->ipif_zoneid = ill->ill_zoneid;
12040
12041     ipif->ipif_refcnt = 0;
12042
12043     if (insert) {
12044         if ((err = ipif_insert(ipif, ire_type != IRE_LOOPBACK)) != 0) {
12045             mi_free(ipif);
12046             if (errorp != NULL)
12047                 *errorp = err;
12048             return (NULL);
12049         }
12050         /* -1 id should have been replaced by real id */
12051         id = ipif->ipif_id;
12052         ASSERT(id >= 0);
12053     }
12054
12055     if (ill->ill_name[0] != '\0')
12056         ipif_assign_seqid(ipif);
12057
12058     /*
12059      * If this is the zeroth ipif on the IPMP ill, create the illgrp
12060      * (which must not exist yet because the zeroth ipif is created once
12061      * per ill). However, do not link it to the ipmp_grp_t until
12062      * I_PLINK is called; see ip_sioctl_plink_ipmp() for details.
12063      */
12064     if (id == 0 && IS_IPMP(ill)) {
12065         if (ipmp_illgrp_create(ill) == NULL) {
12066             if (insert) {
12067                 rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
12068                 ipif_remove(ipif);
12069                 rw_exit(&ipst->ips_ill_g_lock);
12070             }
12071             mi_free(ipif);
12072             if (errorp != NULL)
12073                 *errorp = ENOMEM;

```

```

12074         return (NULL);
12075     }
12076 }
12077
12078 /*
12079  * We grab ill_lock to protect the flag changes. The ipif is still
12080  * not up and can't be looked up until the ioctl completes and the
12081  * IPIF_CHANGING flag is cleared.
12082  */
12083 mutex_enter(&ill->ill_lock);
12084
12085 ipif->ipif_ire_type = ire_type;
12086
12087 if (ipif->ipif_isv6) {
12088     ill->ill_flags |= ILLF_IPV6;
12089 } else {
12090     ipaddr_t inaddr_any = INADDR_ANY;
12091
12092     ill->ill_flags |= ILLF_IPV4;
12093
12094     /* Keep the IN6_IS_ADDR_V4MAPPED assertions happy */
12095     IN6_IPADDR_TO_V4MAPPED(inaddr_any,
12096                           &ipif->ipif_v6lcl_addr);
12097     IN6_IPADDR_TO_V4MAPPED(inaddr_any,
12098                           &ipif->ipif_v6subnet);
12099     IN6_IPADDR_TO_V4MAPPED(inaddr_any,
12100                           &ipif->ipif_v6net_mask);
12101     IN6_IPADDR_TO_V4MAPPED(inaddr_any,
12102                           &ipif->ipif_v6brd_addr);
12103     IN6_IPADDR_TO_V4MAPPED(inaddr_any,
12104                           &ipif->ipif_v6pp_dst_addr);
12105 }
12106
12107 /*
12108  * Don't set the interface flags etc. now, will do it in
12109  * ip_ll_subnet_defaults.
12110  */
12111 if (!initialize)
12112     goto out;
12113
12114 /*
12115  * NOTE: The IPMP meta-interface is special-cased because it starts
12116  * with no underlying interfaces (and thus an unknown broadcast
12117  * address length), but all interfaces that can be placed into an IPMP
12118  * group are required to be broadcast-capable.
12119  */
12120 if (ill->ill_bcast_addr_length != 0 || IS_IPMP(ill)) {
12121     /*
12122      * Later detect lack of DLPI driver multicast capability by
12123      * catching DL_ENABMULTI_REQ errors in ip_rput_dlpi().
12124      */
12125     ill->ill_flags |= ILLF_MULTICAST;
12126     if (!ipif->ipif_isv6)
12127         ipif->ipif_flags |= IPIF_BROADCAST;
12128 } else {
12129     if (ill->ill_net_type != IRE_LOOPBACK) {
12130         if (ipif->ipif_isv6)
12131             /*
12132              * Note: xresolv interfaces will eventually need
12133              * NOARP set here as well, but that will require
12134              * those external resolvers to have some
12135              * knowledge of that flag and act appropriately.
12136              * Not to be changed at present.
12137              */
12138             ill->ill_flags |= ILLF_NONUD;
12139         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     ipldbg(("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     ipldbg(("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         ipldb("couldn't find ncec for ipif %p leaving !ready\n",
12290             (void *)ipif);
12291         return;
12292     }
12293     if (!ncec_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_mthead == NULL && ipx->ipx_mthead == 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     }

```

```

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_ioctl_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 SIOCCLIFGROUPNAME request.
12521  */
12522 /* ARGSUSED */
12523 int
12524 ip_ioctl_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_usersrc_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_ioctl_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_rehold(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_ioctl_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 *lifreq = 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) strcpy(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_ioctl_get_groupname(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
12669     ip_ioctl_cmd_t *pip, void *ifreq)
12670 {
12671     ipmp_grp_t      *grp;
12672     struct lifreq    *lifr = ifreq;
12673     ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;
12674
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) strcpy(lifr->lifr_groupname, grp->gr_name, LIFGRNAMSIZ);
12680     rw_exit(&ipst->ips_ipmp_lock);
12681     return (0);
12682 }
12683
12684 /*
12685  * Process an SIOCGLIFGROUPINFO request.
12686  */
12687 /* ARGSUSED */
12688 int
12689 ip_ioctl_groupinfo(ipif_t *dummy_ipif, sin_t *sin, queue_t *q, mblk_t *mp,
12690     ip_ioctl_cmd_t *pip, void *dummy)
12691 {
12692     ipmp_grp_t      *grp;
12693     lifgroupinfo_t *lifgr;
12694     ip_stack_t      *ipst = CONNQ_TO_IPST(q);
12695
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';
12699
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 }
12709
12710 static void
12711 ill_dl_down(ill_t *ill)
12712 {
12713     DTRACE_PROBE2(ill_downup, char *, "ill_dl_down", ill_t *, ill);
12714
12715     /*
12716      * The ill is down; unbind but stay attached since we're still
12717      * associated with a PPA. If we have negotiated DLPI capabilities
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;
12725
12726     ipldbg(("ill_dl_down(%s)\n", ill->ill_name));
12727
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         ipldbg(("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 }
12764
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;
12771
12772     ASSERT(DB_TYPE(mp) == M_PROTO || DB_TYPE(mp) == M_PCPROTO);
12773
12774     dlp = (union DL_primitives *)mp->b_rptr;
12775     prim = dlp->dl_primitive;
12776
12777     ipldbg(("ill_dlpi_dispatch: sending %s (%u) to %s\n",
12778         dl_primstr(prim), prim, ill->ill_name));
12779
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     }
12793
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 enqueueing.
12845      */
12846     if (!IAM_WRITER_ILL(ill)) {
12847         ill_rehold(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         ipldbg(("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_INVALID) {
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_INVALID);
12995     ASSERT(ill->ill_dlpi_pending == prim);

12997     ipldbg(("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_INVALID;
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.ixas_zoneid = ALL_ZONES;
13188     ixas.ixas_cred = kcred;
13189     ixas.ixas_cpuid = NOPID;
13190     ixas.ixas_tsl = NULL;
13191     /*
13192      * Here we set ixas_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.ixaf_index = ill->ill_phyint->phyint_ifindex;
13197     ixas.ixaf_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->brptr;

13229             ixas.ixaf_multicast_ttl = ip6h->ip6_hops;
13230             ixas.ixaf_flags = IXAF_BASIC_SIMPLE_V6;
13231         } else {
13232             ipha_t *ipha = (ipha_t *)mp->brptr;

13234             ixas.ixaf_multicast_ttl = ipha->ipha_ttl;
13235             ixas.ixaf_flags = IXAF_BASIC_SIMPLE_V4;
13236             ixas.ixaf_flags &= ~IXAF_SET_ULP_CKSUM;
13237         }
13238         ixas.ixaf_flags &= ~IXAF_VERIFY_SOURCE;
13239         ixas.ixaf_flags |= IXAF_MULTICAST_LOOP | IXAF_SET_SOURCE;
13240         (void) ip_output_simple(mp, &ixas);
13241         ixaf_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 ioctl's 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_ioctl_flags
13317 * ipif_down
13318 * wait for ipif to be quiescent
13319 * ipif_down_tail
13320 * ip_ioctl_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     ipldbg(("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_ioctl_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     ipldbg(("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_ioctl_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_ioctl_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     }
13731
13732     if (cp <= name) {
13733         cp = endp;
13734     }
13735     last = *cp;
13736     *cp = '\0';
13737
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);
13749
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_strotol(cp, NULL, 0, &id) != 0) {
13756             ill_refrele(ill);
13757             return (NULL);
13758         }
13759     }
13760
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     }
13788
13789     if (!do_alloc) {

```

```

13790         mutex_exit(&ill->ill_lock);
13791         ill_refrele(ill);
13792         return (NULL);
13793     }
13794
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 overridden 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 }
13817
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;
13834
13835     if (error != NULL)
13836         *error = 0;
13837
13838     if (namelen == 0) {
13839         if (error != NULL)
13840             *error = ENXIO;
13841         return (NULL);
13842     }
13843
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     }
13850
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 }
13871
13872 if (cp <= name) {
13873     cp = endp;
13874 } else {
13875     *cp = '\0';
13876 }
13877
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);
13889
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_strotol(cp, NULL, 0, &id) != 0) {
13896         ill_refrele(ill);
13897         if (error != NULL)
13898             *error = ENXIO;
13899         return (NULL);
13900     }
13901 }
13902
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         }
13918     }
13919     if (!(IPIF_IS_CHANGING(ipif) ||
13920         IPIF_IS_CONDEMNED(ipif) ||
13921         IAM_WRITER_IPIF(ipif))) {

```

```

13922         ipif_rehold_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_enqueue(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 }
13956
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     icmp_h_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;
13973
13974 #define REPLY_LEN      (sizeof (icmp_ipha) + sizeof (icmp_h_t) + IP_ADDR_LEN)
13975
13976     if (!ipst->ips_ip_respond_to_address_mask_broadcast)
13977         return;
13978
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);
13983
13984     if (ipif->ipif_lcl_addr == INADDR_ANY)
13985         return;
13986
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.ixas_flags = IXAF_BASIC_SIMPLE_V4;
14008     ixas.ixas_zoneid = ALL_ZONES;
14009     ixas.ixas_ifindex = 0;
14010     ixas.ixas_ipst = ipst;
14011     ixas.ixas_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     ipldbg(("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         ipldbg(("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     ipldbg(("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));
14126
14127     ipldbg(("ipif_multicast_down\n"));
14128
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 }
14138
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;
14148
14149     ipldbg(("ill_recover_saved_ire(%s)", ill->ill_name));
14150
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;
14155
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     }
14187
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     }
14196
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;
14210
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         ipldbg(("ill_recover_saved_ire: FAILED\n"));
14222     } else if (nire != ire) {
14223         ipldbg(("ill_recover_saved_ire: duplicate ire %p\n",
14224             (void *)nire));
14225         ire_delete(nire);
14226     } else {
14227         ipldbg(("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 }
14236
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));
14245
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     ipldb(("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 defereneces 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      rw_exit(&ipst->ips_ill_g_lock);
14465
14466      /*
14467       * Tell SCTP that the ipif has moved. Note that even if we
14468       * had to allocate a new ipif, the original sequence id was
14469       * preserved and therefore SCTP won't know.
14470       */
14471      sctp_move_ipif(ipif, ill, ipmp_ill);
14472
14473      /*
14474       * If the ipif being brought up was on slot zero, then we
14475       * first need to bring up the placeholder we stuck there. In
14476       * ip_rput_dlpi_writer(), arp_bringup_done(), or the recursive
14477       * call to ipif_up() itself, if we successfully bring up the
14478       * placeholder, we'll check ill_move_ipif and bring it up too.
14479       */
14480      if (ipif_orig_id == 0) {
14481          ASSERT(ill->ill_move_ipif == NULL);
14482          ill->ill_move_ipif = ipif;
14483          if ((err = ipif_up(ill->ill_ipif, q, mp)) == 0)
14484              ASSERT(ill->ill_move_ipif == NULL);
14485          if (err != EINPROGRESS)
14486              ill->ill_move_ipif = NULL;
14487          return (err);
14488      }
14489
14490      /*
14491       * Bring it up on the IPMP ill.
14492       */
14493      return (ipif_up(ipif, q, mp));
14494  }
14495
14496      /* Skip arp/ndp for any loopback interface. */
14497      if (ill->ill_wq != NULL) {
14498          conn_t *connp = CONN_Q(q) ? Q_TO_CONN(q) : NULL;
14499          ipsq_t *ipsq = ill->ill_phyint->phyint_ipsq;
14500
14501          if (!ill->ill_dl_up) {
14502              /*
14503               * ill_dl_up is not yet set. i.e. we are yet to
14504               * DL_BIND with the driver and this is the first
14505               * logical interface on the ill to become "up".
14506               * Tell the driver to get going (via DL_BIND REQ).
14507               * Note that changing "significant" IFF flags
14508               * address/netmask etc cause a down/up dance, but
14509               * does not cause an unbind (DL_UNBIND) with the driver
14510               */
14511              return (ill_dl_up(ill, ipif, mp, q));
14512          }
14513      }
14514      /*

```

```

14516     * ipif_resolver_up may end up needing to bind/attach
14517     * the ARP stream, which in turn necessitates a
14518     * DLPI message exchange with the driver. ioctl's 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     ipldbg(("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 ioctl's 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 ioctl's 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 ioctl's 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     ipldbg(("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     ipldbg(("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
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
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_PUBLISHER;
14800
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
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
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
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
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
14877     /* Join the allhosts multicast address */
14878     ipif_multicast_up(ipif);
14880
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
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
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         **irepl;
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         ipldbg(("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             ipldbg(("ipif_up_done: NULL ire_local\n"));
14963             err = ENOMEM;
14964             goto bad;
14965         }
14966     } else {
14967         ipldbg((
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         ipldbg(("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             ipldbg(("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 (irepl = irep; irepl > ire_array; ) {
15036         irepl--;
15037         if (*irepl == NULL) {
15038             ipldbg(("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 (irepl = irep; irepl > ire_array; ) {
15117      irepl--;
15118      ASSERT(!MUTEX_HELD(&((*irepl)->ire_ill->ill_lock));
15119      *irepl = ire_add(*irepl);
15120      if (*irepl == NULL) {
15121          err = ENOMEM;
15122          goto bad2;
15123      }
15124  }

15126  for (irepl = irep; irepl > ire_array; ) {
15127      irepl--;
15128      /* refheld by ire_add. */
15129      if (*irepl != NULL) {
15130          ire_refrele(*irepl);
15131          *irepl = 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;

15144          ire = ire_ftable_lookup_v4(ipif->ipif_brd_addr, 0, 0,
15145          IRE_BROADCAST, ipif->ipif_ill, ALL_ZONES, NULL,
15146          (MATCH_IRE_TYPE | MATCH_IRE_ILL), 0, ipst, NULL);

15147          if (ire == NULL) {
15148              /*
15149              * If there isn't a matching broadcast IRE,
15150              * revert to the default for this netmask.
15151              */
15152              ipif->ipif_v6brd_addr = ipv6_all_zeros;
15153              mutex_enter(&ipif->ipif_ill->ill_lock);
15154              ipif_set_default(ipif);
15155              mutex_exit(&ipif->ipif_ill->ill_lock);
15156          } else {
15157              ire_refrele(ire);
15158          }
15159      }
15160  }

15161  }

15163  return (0);
15164

15166 bad2:
15167  ill->ill_ipif_up_count--;
15168  ipif->ipif_flags &= ~IPIF_UP;

15170 bad:
15171  ipldbg(("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 lose 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 availability 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_usersrc, boolean_t *notready)
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_rhnp, *dst_rhnp;
15348     ip_stack_t *ipst = ill->ill_ipst;
15349     boolean_t samenet;

15351     if (ill->ill_usersrc_ifindex != 0 && allow_usersrc) {
15352         usill = ill_lookup_on_ifindex(ill->ill_usersrc_ifindex,
15353             B_FALSE, ipst);
15354         if (usill != NULL)
15355             ill = usill; /* Select source from usersrc 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_rhnp = NULL;
15378     if (is_system_labeled()) {
15379         dst_rhnp = find_tpc(&dst, IPV4_VERSION, B_FALSE);
15380         if (dst_rhnp == NULL)
15381             return (NULL);
15382         if (dst_rhnp->tpc_tp.host_type != UNLABELED) {
15383             TPC_RELE(dst_rhnp);
15384             dst_rhnp = 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     * ill_s 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 (notready != NULL)
15430                 *notready = 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_rhttp != NULL) {
15452     boolean_t incompat;

15454     src_rhttp = find_tpc(&ipif->ipif_lcl_addr,
15455                         IPV4_VERSION, B_FALSE);
15456     if (src_rhttp == NULL)
15457         continue;
15458     incompat = src_rhttp->tpc_tp.host_type != SUN_CIPSO ||
15459               src_rhttp->tpc_tp.tp_doi !=
15460               dst_rhttp->tpc_tp.tp_doi ||
15461               (!blinrange(&dst_rhttp->tpc_tp.tp_def_label,
15462                           &src_rhttp->tpc_tp.tp_sl_range_cipso) &&
15463                !blinlset(&dst_rhttp->tpc_tp.tp_def_label,
15464                          src_rhttp->tpc_tp.tp_sl_set_cipso));
15465     TPC_RELE(src_rhttp);
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_rhttp != NULL)
15522     TPC_RELE(dst_rhttp);

15524 #ifdef DEBUG
15525     if (ipif == NULL) {
15526         char buf1[INET6_ADDRSTRLEN];

15528         ipldbg(("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         ipldbg(("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 ussrc is set, swap the ill to be the ussrc 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 *srpc,
15579 uint32_t *generation, uint64_t *flagsp)
15580 {
15581     ipif_t *ipif;
15582     boolean_t notready = B_FALSE; /* Set if !ipif_addr_ready found */
15583
15584     if (flagsp != NULL)
15585         *flagsp = 0;
15586
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     }
15597
15598     if (CLASSD(dst) && multicast_ifaddr != INADDR_ANY) {
15599         *srpc = multicast_ifaddr;
15600         return (0);
15601     }
15602
15603     /* Was RTF_SETSRC set on the first IRE in the recursive lookup? */
15604     if (setsrc != INADDR_ANY) {
15605         *srpc = 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     *srpc = ipif->ipif_lcl_addr;
15616     if (flagsp != NULL)
15617         *flagsp = ipif->ipif_flags;
15618     ipif_refrele(ipif);
15619     return (0);
15620 }
15621
15622 /* ARGSUSED */
15623 int
15624 if_unitssel_restart(ipif_t *ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
15625 ip_ioctl_cmd_t *pip, 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 }
15635
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_unitssel(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
15643 ip_ioctl_cmd_t *pip, 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;
15649
15650     if (q->q_next == NULL) {
15651         ipldbg(("if_unitssel: IF_UNITSEL: no q_next\n"));
15652         return (EINVAL);
15653     }
15654
15655     if (((ill_t *)q->q_ptr)->ill_name[0] != '\0')
15656         return (EALREADY);
15657
15658     do {
15659         q1 = q1->q_next;
15660     } while (q1->q_next);
15661     cp = q1->q_qinfo->q1_minfo->mi_idname;
15662     (void) sprintf(interf_name, "%s%d", cp, ppa);
15663
15664     /*
15665      * Here we are not going to delay the ioack until after
15666      * ACKs from DL_ATTACH_REQ/DL_BIND_REQ. So no need to save the
15667      * original ioctl message before sending the requests.
15668      */
15669     return (ipif_set_values(q, mp, interf_name, &ppa));
15670 }
15671
15672 /* ARGSUSED */
15673 int
15674 ip_siocctl_sifname(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
15675 ip_ioctl_cmd_t *pip, void *dummy_ifreq)
15676 {
15677     return (ENXIO);
15678 }
15679
15680 /*
15681  * Create any IRE_BROADCAST entries for 'ipif', and store those entries in
15682  * 'irep'. Returns a pointer to the next free 'irep' entry
15683  * A mirror exists in ipif_delete_bcast_ires().
15684  * The management of any "extra" or seemingly duplicate IRE_BROADCASTs is
15685  * done in ire_add.
15686  */
15687 static ire_t **
15688 ipif_create_bcast_ires(ipif_t *ipif, ire_t **irep)
15689 {
15690     ipaddr_t addr;
15691     ipaddr_t netmask = ip_net_mask(ipif->ipif_lcl_addr);
15692     ipaddr_t subnetmask = ipif->ipif_net_mask;
15693     ill_t *ill = ipif->ipif_ill;
15694     zoneid_t zoneid = ipif->ipif_zoneid;
15695
15696     ipldbg(("ipif_create_bcast_ires: creating broadcast IRES\n"));
15697
15698     ASSERT(ipif->ipif_flags & IPIF_BROADCAST);
15699     ASSERT(!(ipif->ipif_flags & IPIF_NOXMIT));
15700
15701     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 queuing 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_ioctl_slifname(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
15817 ip_ioctl_cmd_t *pip, void *if_req)
15818 {
15819     ill_t        *ill = q->q_ptr;
15820     phyint_t     *phyi;
15821     ip_stack_t   *ipst;
15822     struct lifreq *lifreq = if_req;
15823     uint64_t     new_flags;

15825     ASSERT(ipif != NULL);
15826     ipldbg(("ip_ioctl_slifname %s\n", lifreq->lifr_name));

15828     if (q->q_next == NULL) {
15829         ipldbg(("if_ioctl_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             ipldb("if_ioctl_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     * Sanity 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         ipldb("ip_ioctl_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         ipldb("ip_ioctl_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_ioctl_slifname_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
15924                          ip_ioctl_cmd_t *pip, 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_ioctl_slifindex(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
15973 ip_ioctl_cmd_t *pip, 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_ioctl_get_lifindex(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16048 ip_ioctl_cmd_t *pip, void *ifreq)
16049 {
16050     struct ifreq  *ifr = (struct ifreq *)ifreq;
16051     struct lifreq *lifr = (struct lifreq *)ifreq;

16053     ipldbg(("ip_ioctl_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_ioctl_get_lifzone(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16067 ip_ioctl_cmd_t *pip, void *ifreq)
16068 {
16069     struct lifreq *lifr = (struct lifreq *)ifreq;

16071     ipldbg(("ip_ioctl_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_ioctl_slifzone(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16085 ip_ioctl_cmd_t *pip, 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      ipldbg(("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 *liffr = (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 = liffr->lifr_zoneid) == ALL_ZONES)
16199          zoneid = GLOBAL_ZONEID;

16201      ipldbg(("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 SIOCGLIFZONE 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 SIOCGLIFZONE 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_ioctl_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_ioctl_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     ipldbg(("ip_ioctl_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     ipldbg(("ip_ioctl_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_ioctl_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         ipldbg(("ip_ioctl_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_ioctl_get_dadstate(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
16540 ip_ioctl_cmd_t *pip, 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) strncpy(ill->ill_phyint->phyint_name, ill->ill_name, LIFNAMSTZ);

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     } else {
16766         ill->ill_mcast_type = ipst->ips_mld_max_version;
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 *mpl, *mp2;
16801     struct iocblk *iocp;
16802     struct lifreq *lifr;

16804     mpl = mkioch(SIOCSLIFNAME);
16805     if (mpl == NULL)
16806         return;
16807     mp2 = alloch(sizeof (struct lifreq), BPRI_HI);
16808     if (mp2 == NULL) {
16809         freeb(mpl);
16810         return;
16811     }

16813     mpl->b_cont = mp2;
16814     iocp = (struct iocblk *)mpl->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, mpl);
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     ipldbg(("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 phynt. 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-multicast 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_fhtable_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_nodet_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 };
17433
17434     ASSERT(IS_IPMP(ill));
17435
17436     (void) ddi_stirtoul(hw_serial, NULL, 10, &hostid);
17437     ic.ic_hostid = htonl((uint32_t)hostid);
17438
17439     (void) strncpy(ic.ic_ifname, ill->ill_name, LIFNAMSIZ);
17440
17441     if ((zp = zone_find_by_id(ill->ill_zoneid)) != NULL) {
17442         (void) strncpy(ic.ic_zonename, zp->zone_name, ZONENAME_MAX);
17443         zone_rele(zp);
17444     }
17445
17446     MD5Init(&ctx);
17447     MD5Update(&ctx, &ic, sizeof(ic));
17448     MD5Final(hash, &ctx);
17449
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 }
17457
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;
17465
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_mbcst_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 }
17483
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;
17491
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_mbcst_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 }
17509
17510 /* ARGSUSED */
17511 static void
17512 ip_mbcst_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;
17520
17521     dlur = (dl_unitdata_req_t *)ill->ill_bcast_mp->b_rprtr;
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     }
17529
17530     bcopy(bphys_addr, m_physaddr, ill->ill_phys_addr_length);
17531 }
17532
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;
17540
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, 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->d1_dest_addr_offset;
17596     } else {
17597         bphys_addr = (uchar_t *)dlur + dlur->d1_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, 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->d1_dest_addr_offset;
17627     } else {
17628         bphys_addr = (uchar_t *)dlur + dlur->d1_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_genfv6intfid(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
17703     ill = ill_lookup_on_ifindex(index, isv6, ipst);
17704     if (ill == NULL)
17705         return (NULL);
17706
17707     mutex_enter(&ill->ill_lock);
17708     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
17709         if (IPIF_IS_CONDEMNED(ipif))
17710             continue;
17711         if (zoneid != ALL_ZONES && ipif->ipif_zoneid != zoneid &&
17712             ipif->ipif_zoneid != ALL_ZONES)
17713             continue;
17714
17715         mutex_exit(&ill->ill_lock);
17716         return (ill);
17717     }
17718     mutex_exit(&ill->ill_lock);
17719     ill_refrele(ill);
17720     return (NULL);
17722 }
17723 /*
17724  * Return a pointer to an ipif_t given a combination of (ill_idx, ipif_id)
17725  * If a pointer to an ipif_t is returned then the caller will need to do
17726  * an ill_refrele().
17727  */
17728 ipif_t *
17729 ipif_getby_indexes(uint_t ifindex, uint_t lifidx, boolean_t isv6,
17730     ip_stack_t *ipst)
17731 {
17732     ipif_t *ipif;
17733     ill_t *ill;
17734
17735     ill = ill_lookup_on_ifindex(ifindex, isv6, ipst);
17736     if (ill == NULL)
17737         return (NULL);
17738
17739     mutex_enter(&ill->ill_lock);
17740     if (ill->ill_state_flags & ILL_CONDEMNED) {
17741         mutex_exit(&ill->ill_lock);
17742         ill_refrele(ill);
17743         return (NULL);
17744     }
17745
17746     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
17747         if (!IPIF_CAN_LOOKUP(ipif))
17748             continue;
17749         if (lifidx == ipif->ipif_id) {
17750             ipif_refhold_locked(ipif);

```

```

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
17770     if (ill->ill_isv6) {
17771         if (is_system_labeled())
17772             ill->ill_inputfn = ill_input_full_v6;
17773         else
17774             ill->ill_inputfn = ill_input_short_v6;
17775     } else {
17776         if (is_system_labeled())
17777             ill->ill_inputfn = ill_input_full_v4;
17778         else if (ill->ill_dhcpinit != 0)
17779             ill->ill_inputfn = ill_input_full_v4;
17780         else if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_RSVP].connf_head
17781             != NULL)
17782             ill->ill_inputfn = ill_input_full_v4;
17783         else if (ipst->ips_ip_cgtp_filter &&
17784             ipst->ips_ip_cgtp_filter_ops != NULL)
17785             ill->ill_inputfn = ill_input_full_v4;
17786         else
17787             ill->ill_inputfn = ill_input_short_v4;
17788     }
17790 }
17791 /*
17792  * Re-evaluate ill_inputfn for all the IPv4 ill.
17793  * Used when RSVP and CGTP comes and goes.
17794  */
17795 void
17796 ill_set_inputfn_all(ip_stack_t *ipst)
17797 {
17798     ill_walk_context_t ctx;
17799     ill_t *ill;
17800
17801     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
17802     ill = ILL_START_WALK_V4(&ctx, ipst);
17803     for (; ill != NULL; ill = ill_next(&ctx, ill))
17804         ill_set_inputfn(ill);
17805
17806     rw_exit(&ipst->ips_ill_g_lock);
17808 }
17809 /*
17810  * Set the physical address information for 'ill' to the contents of the
17811  * dl_notify_ind_t pointed to by 'mp'. Must be called as writer, and will be
17812  * asynchronous if 'ill' cannot immediately be quiesced -- in which case
17813  * EINPROGRESS will be returned.
17814  */
17815 int
17816 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_dipi_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_lls 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_lls_mp);
18001     ill->ill_nd_lls = ndmp->b_rptr + addroff;
18002     ill->ill_nd_lls_mp = ndmp;
18003     ill->ill_nd_lls_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_dipi_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_timeout = 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 'bufsize' 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, SIOCCLIFREMOVEIF, &lifr, sizeof (lifr), cr);
18175      if (err != 0 && err != ENXIO) {
18176          ip0dbg(("ip_loopback_removeif: IP%s SIOCCLIFREMOVEIF 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, SIOCGLIFGROUPNAME, 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(ddd_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_dataalen = 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_dataalen = 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_dataalen);
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_UNSQL_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         ipldbg(("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         ipldbg(("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 *mpl;
18575     ipif_t *ipif;
18576     conn_t *connp = NULL;
18577     ipsq_t *ipsq;
18578     queue_t *q;

18580     ipldbg(("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     mpl = ipsq_pending_mp_get(ipsq, &connp);
18587     ASSERT(!((mpl != NULL) ^ (ipif != NULL)));
18588     if (mpl == 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, mpl);
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     ipldbg(("bringing up ipif %s 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, mpl);
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, mpl, 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 *mpl;
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     mpl = ipsq_pending_mp_get(ipsq, &connp);
18679     ASSERT(!((mpl != NULL) ^ (ipif != NULL)));
18680     if (mpl == 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, mpl);
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, mpl, 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 shouldn't 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_ioctl_ilb_cmd(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
18742 ip_ioctl_cmd_t *pip, 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_CONNQ(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->nice_ipif_cnt == 0)
19006                 ncec_delete(nce->nice_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_usersrc(ill_t *usill)
19040 {
19041     ip_stack_t *ipst = usill->ill_ipst;
19042     ill_t *ill;

19044     ASSERT(usill != NULL);

19046     /* ill_g_usersrc_lock protects ill_usersrc_grp_next */
19047     rw_enter(&ipst->ips_ill_g_usersrc_lock, RW_WRITER);
19048     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
19049     for (ill = usill->ill_usersrc_grp_next; ill != NULL && ill != usill;
19050         ill = ill->ill_usersrc_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_usersrc_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  * EPNOSUPPORT - 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 *pip, void *if_req)
19086 {
19087     struct sockaddr *sock;
19088     struct ifreq *ifr;
19089     mblk_t *mpl;
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     ipldbg(("ip_sioctl_get_hwaddr(%)\\n", ill->ill_name));

19104     /* Existence of mpl has been checked in ip_wput_nondata */
19105     mpl = mp->b_cont->b_cont;
19106     ifr = (struct ifreq *)mpl->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 expectation 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 *pip, void *if_req)
19132 {
19133     struct sockaddr_dl *sock;
19134     struct lifreq *lifr;
19135     mblk_t *mpl;

```

```
19136     ill_t *ill;
19137
19138     ASSERT(ipif != NULL);
19139     ill = ipif->ipif_ill;
19140
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     }
19147
19148     ipldbg(("ip_siocctl_get_lifhwaddr(%s)\n", ill->ill_name));
19149
19150     /* Existence of mpl has been checked in ip_wput_nondata */
19151     mpl = mp->b_cont->b_cont;
19152     lifr = (struct lifreq *)mpl->b_rptr;
19153
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);
19169
19170     return (0);
19171 }
```

new/usr/src/uts/common/inet/ip/ip_input.c

1

```
*****
89485 Wed Jun 13 12:04:55 2012
new/usr/src/uts/common/inet/ip/ip_input.c
%B
*****
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) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
24  *
25  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
26  */
27 /* Copyright (c) 1990 Mentat Inc. */

29 #include <sys/types.h>
30 #include <sys/stream.h>
31 #include <sys/dlpi.h>
32 #include <sys/stropts.h>
33 #include <sys/sysmacros.h>
34 #include <sys/strsubr.h>
35 #include <sys/strlog.h>
36 #include <sys/strsun.h>
37 #include <sys/zone.h>
38 #define _SUN_TPI_VERSION 2
39 #include <sys/tihdr.h>
40 #include <sys/xti_inet.h>
41 #include <sys/ddi.h>
42 #include <sys/sunddi.h>
43 #include <sys/cmn_err.h>
44 #include <sys/debug.h>
45 #include <sys/kobj.h>
46 #include <sys/modctl.h>
47 #include <sys/atomic.h>
48 #include <sys/policy.h>
49 #include <sys/priv.h>

51 #include <sys/system.h>
52 #include <sys/param.h>
53 #include <sys/kmem.h>
54 #include <sys/sdt.h>
55 #include <sys/socket.h>
56 #include <sys/vtrace.h>
57 #include <sys/isa_defs.h>
58 #include <sys/mac.h>
59 #include <net/if.h>
60 #include <net/if_arp.h>
61 #include <net/route.h>
```

new/usr/src/uts/common/inet/ip/ip_input.c

2

```
62 #include <sys/sockio.h>
63 #include <netinet/in.h>
64 #include <net/if_dl.h>

66 #include <inet/common.h>
67 #include <inet/mi.h>
68 #include <inet/mib2.h>
69 #include <inet/nd.h>
70 #include <inet/arp.h>
71 #include <inet/snmpcom.h>
72 #include <inet/kstatcom.h>

74 #include <netinet/igmp_var.h>
75 #include <netinet/ip6.h>
76 #include <netinet/icmp6.h>
77 #include <netinet/sctp.h>

79 #include <inet/ip.h>
80 #include <inet/ip_impl.h>
81 #include <inet/ip6.h>
82 #include <inet/ip6_asp.h>
83 #include <inet/optcom.h>
84 #include <inet/tcp.h>
85 #include <inet/tcp_impl.h>
86 #include <inet/ip_multi.h>
87 #include <inet/ip_if.h>
88 #include <inet/ip_ire.h>
89 #include <inet/ip_ftable.h>
90 #include <inet/ip_rts.h>
91 #include <inet/ip_ndp.h>
92 #include <inet/ip_listutils.h>
93 #include <netinet/igmp.h>
94 #include <netinet/ip_mroute.h>
95 #include <inet/ipp_common.h>

97 #include <net/pfkeyv2.h>
98 #include <inet/sadb.h>
99 #include <inet/ipsec_impl.h>
100 #include <inet/ipdrop.h>
101 #include <inet/ip_netinfo.h>
102 #include <inet/ilb_ip.h>
103 #include <sys/queue_impl.h>
104 #include <sys/queue.h>

106 #include <sys/ethernet.h>
107 #include <net/if_types.h>
108 #include <sys/cpuvar.h>

110 #include <ipp/ipp.h>
111 #include <ipp/ipp_impl.h>
112 #include <ipp/ipgpc/igppc.h>

114 #include <sys/pattn.h>
115 #include <inet/ipclassifier.h>
116 #include <inet/sctp_ip.h>
117 #include <inet/sctp/sctp_impl.h>
118 #include <inet/udp_impl.h>
119 #include <inet/dccp/dccp_impl.h>
120 #endif /* ! codereview */
121 #include <sys/sunddi.h>

123 #include <sys/tsol/label.h>
124 #include <sys/tsol/tnet.h>

126 #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_rcv_attr_t *);

135 static void ip_input_broadcast_v4(ire_t *, mblk_t *, ipha_t *,
136 ip_rcv_attr_t *);
137 static void ip_input_multicast_v4(ire_t *, mblk_t *, ipha_t *,
138 ip_rcv_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_rcv_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_rcv_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_rcv_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 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_cpuid = 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_rcv_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 }
407
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 }
415
416 return (NULL);
417 }
418
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_rcv_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            cgtpflt_pkt;
435
436     ASSERT(ira->ira_tsl == NULL);
437
438     /*
439     * Attach any necessary label information to
440     * this packet
441     */
442     if (is_system_labeled()) {
443         ira->ira_flags |= IRAF_SYSTEM_LABELED;
444
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. */
456
457         /* tsol_get_pkt_label sometimes does pullupmsg */

```

```

458         ipha = (ipha_t *)mp->b_rptr;
459     }
460
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     cgtpflt_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;
471
472         stackid = ipst->ips_netstack->netstack_stackid;
473         /*
474         * CGTP and IPMP are mutually exclusive so
475         * phyint_ifindex is fine here.
476         */
477         cgtpflt_pkt =
478             ipst->ips_ip_cgtp_filter_ops->cfo_filter(stackid,
479             ill->ill_phyint->phyint_ifindex, mp);
480         if (cgtpflt_pkt == CGTP_IP_PKT_DUPLICATE) {
481             ip_drop_input("CGTP_IP_PKT_DUPLICATE", mp, ill);
482             freemsg(mp);
483             return;
484         }
485     }
486
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;
504
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];
514
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_rcv_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_rcv_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(nextthop, 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, nextthop, 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      nextthop = 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          nextthop = ip_input_options(ipha, nextthop, 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(nextthop)) {
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_ts1, 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_rcv_local_v4 - for IRE_LOCAL
823 * ire_rcv_loopback_v4 - for IRE_LOOPBACK
824 * ire_rcv_multirt_v4 - if RTF_MULTIRT
825 * ire_rcv_noroute_v4 - if RTF_REJECT or RTF_BLAHOLE
826 * ire_rcv_multicast_v4 - for IRE_MULTICAST
827 * ire_rcv_broadcast_v4 - for IRE_BROADCAST
828 * ire_rcv_noaccept_v4 - for ire_noaccept ones
829 * ire_rcv_forward_v4 - for the rest.
830 */
831 (*ire->ire_rcvfn)(ire, mp, ipha, ira);
832 }
833 #undef rptr
835 /*
836 * ire_rcvfn for IREs that need forwarding
837 */
838 void
839 ire_rcv_forward_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_rcv_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 rehold 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_rehold(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_rcvfn)(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_rcv_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_rprtr;
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_LABELLED) {
1062     mblk_t *mpl;
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 ((mpl = 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 = mpl;
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_rcv_attr_t *ira, uint32_t mtu, uint32_t added_tx_len)
1119 {
1120     ill_t *dst_ill = nce->nice_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 iaflags;

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_rcv_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     iaflags = 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_LABELLED) {
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
1320     /* Tag for higher-level protocols */
1321     ira->ira_flags |= IRAF_BROADCAST;
1322
1323     /*
1324     * Whether local or directed broadcast forwarding: don't allow
1325     * for TCP.
1326     */
1327     if (ipha->ipha_protocol == IPPROTO_TCP) {
1328         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1329         ip_drop_input("ipIfStatsInDiscards", mp, ill);
1330         freemsg(mp);
1331         return;
1332     }
1333
1334     /*
1335     * So that we don't end up with dups, only one ill an IPMP group is
1336     * nominated to receive broadcast traffic.
1337     * If we have no cast_ill we are liberal and accept everything.
1338     */
1339     if (IS_UNDER_IPMP(ill)) {
1340         /* For an under ill_grp can change under lock */
1341         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1342         if (!ill->ill_nom_cast && ill->ill_grp != NULL &&
1343             ill->ill_grp->ig_cast_ill != NULL) {
1344             rw_exit(&ipst->ips_ill_g_lock);
1345             /* No MIB since this is normal operation */
1346             ip_drop_input("not nom_cast", mp, ill);
1347             freemsg(mp);
1348             return;
1349         }
1350         ira->ira_ruifindex = ill_get_upper_ifindex(ill);
1351     }
1352
1353     /*
1354     * After reassembly and IPsec we will need to duplicate the
1355     * broadcast packet for all matching zones on the ill.
1356     */
1357     ira->ira_zoneid = ALL_ZONES;
1358
1359     /*
1360     * Check for directed broadcast i.e. ire->ire_ill is different than
1361     * the incoming ill.
1362     * The same broadcast address can be assigned to multiple interfaces
1363     * so have to check explicitly for that case by looking up the alt_ire
1364     */
1365     if (dst_ill == ill && !(ire->ire_flags & RTF_MULTIRT)) {
1366         /* Reassemble on the ill on which the packet arrived */
1367         ip_input_local_v4(ire, mp, ipha, ira);
1368         /* Restore */
1369         ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;
1370         return;
1371     }
1372
1373     /* Is there an IRE_BROADCAST on the incoming ill? */
1374     ipha_dst = ((ira->ira_flags & IRAF_DHCP_UNICAST) ? INADDR_BROADCAST :
1375         ipha->ipha_dst);
1376     alt_ire = ire_ftable_lookup_v4(ipha_dst, 0, 0, IRE_BROADCAST, ill,
1377         ALL_ZONES, ira->ira_tsl,
1378         MATCH_IRE_TYPE|MATCH_IRE_ILL|MATCH_IRE_SECATTR, 0, ipst, NULL);
1379     if (alt_ire != NULL) {
1380         /* Not a directed broadcast */
1381

```

```

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 *irel;
1393
1394         irb = ire->ire_bucket;
1395         irb_refhold(irb);
1396         for (irel = irb->irb_ire; irel != NULL;
1397             irel = irel->ire_next) {
1398             if (IRE_IS_CONDEMNED(irel))
1399                 continue;
1400             if (!(irel->ire_type & IRE_BROADCAST) ||
1401                 (irel->ire_flags & RTF_MULTIRT))
1402                 continue;
1403             ill = irel->ire_ill;
1404             ill_refhold(ill);
1405             break;
1406         }
1407         irb_refrele(irb);
1408         if (irel != NULL) {
1409             ill_t *orig_ill = ira->ira_ill;
1410
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 }
1428
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     }
1451
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;
1457
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);
1464
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;
1472
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     }
1480
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 }
1487
1488 /*
1489 * ire_rcvfn for IRE_MULTICAST.
1490 */
1491 void
1492 ire_rcv_multicast_v4(ire_t *ire, mblk_t *mp, void *iph_arg,
1493     ip_rcv_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;
1498
1499     ASSERT(ire->ire_ill == ira->ira_ill);
1500
1501     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInMcastPkts);
1502     UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCInMcastOctets, ira->ira_pktlen);
1503
1504     /* RSVP hook */
1505     if (ira->ira_flags & IRAF_RSVP)
1506         goto forus;
1507
1508     /* Tag for higher-level protocols */
1509     ira->ira_flags |= IRAF_MULTICAST;
1510
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;
1518
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 }
1543
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 * mrouterd has not joined the allmulti group on this interface.
1549 */
1550 if (ipst->ips_ip_g_mrouter) {
1551     int retval;
1552
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;
1558
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 */
1568
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 }
1601
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_rcv_multicast_v4: multicast for us: 0x%x\n",
1621     ntohl(ipha->ipha_dst)));
1622
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;
1628
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 }
1638
1639 /*
1640 * ire_rcvfn for IRE_OFFLINK with RTF_MULTIRT.
1641 * Drop packets since we don't forward out multirt routes.
1642 */
1643 /* ARGSUSED */
1644 void
1645 ire_rcv_multirt_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_rcv_attr_t *ira)

```

```

1646 {
1647     ill_t         *ill = ira->ira_ill;
1648
1649     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInNoRoutes);
1650     ip_drop_input("Not forwarding out MULTIRT", mp, ill);
1651     freemsg(mp);
1652 }
1653
1654 /*
1655 * ire_rcvfn 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_rcv_loopback_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_rcv_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;
1666
1667     ira->ira_zoneid = GLOBAL_ZONEID;
1668
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;
1678
1679         ip_input_local_v4(ire, mp, ipha, ira);
1680
1681         /* Restore */
1682         ASSERT(ira->ira_ill == ire_ill);
1683         ira->ira_ill = ill;
1684         return;
1685     }
1686
1687     ip_input_local_v4(ire, mp, ipha, ira);
1688 }
1689
1690 /*
1691 * ire_rcvfn for IRE_LOCAL.
1692 */
1693 void
1694 ire_rcv_local_v4(ire_t *ire, mblk_t *mp, void *iph_arg, ip_rcv_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;
1699
1700     /* Make a note for DAD that this address is in use */
1701     ire->ire_last_used_time = LBOLT_FASTPATH;
1702
1703     /* Only target the IRE_LOCAL with the right zoneid. */
1704     ira->ira_zoneid = ire->ire_zoneid;
1705
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_rprtr;
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          *irel;

```

```

1844     mblk_t      *mpl;
1845     ipha_t      *iphal;
1846     uint_t      ira_pktlen = ira->ira_pktlen;
1847     uint16_t    ira_ip_hdr_length = ira->ira_ip_hdr_length;
1849
1849     irb = ire->ire_bucket;
1851
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
1859         ip_fanout_v4(mp, ipha, ira);
1860         return;
1861     }
1862     irb_refhold(irb);
1863     for (irel = irb->irb_ire; irel != NULL; irel = irel->ire_next) {
1864         /* We do the main IRE after the end of the loop */
1865         if (irel == ire)
1866             continue;
1868
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(irel->ire_addr == ire->ire_addr);
1874         if (!(irel->ire_type & IRE_BROADCAST))
1875             continue;
1877
1877         if (IRE_IS_CONDEMNED(irel))
1878             continue;
1880
1880         mpl = copymsg(mp);
1881         if (mpl == 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 = irel->ire_zoneid;
1888         iphal = (ipha_t *)mpl->b_rptr;
1889         ip_fanout_v4(mpl, iphal, ira);
1890
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_rcv_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      *mpl;
1925         ipha_t      *iphal;
1926         uint_t      ira_pktlen = ira->ira_pktlen;
1927         uint16_t    ira_ip_hdr_length = ira->ira_ip_hdr_length;
1929
1929         /* ire_rcv_multicast has switched to the upper ill for IPMP */
1930         ASSERT(!IS_UNDER_IPMP(ill));
1932
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
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
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
1971             ira->ira_zoneid = zoneid;
1973
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;
1977
1978         mpl = copymsg(mp);
1979         if (mpl == 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 *)mpl->b_rptr;
1986         ip_fanout_v4(mpl, 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     }
1994
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 }
2006
2007
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_rcv_attr_t *ira)
2015 {
2016     uint16_t    *up;
2017     uint16_t    lport;
2018     zoneid_t    zoneid;
2019
2020     ASSERT(ira->ira_flags & IRAF_SYSTEM_LABELED);
2021
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;
2030
2031     if (ira->ira_zoneid != ALL_ZONES)
2032         return;
2033
2034     ira->ira_flags |= IRAF_TX_SHARED_ADDR;
2035
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);
2043
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];
2050
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 }
2078
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;
2086
2087     switch (protocol) {
2088     case IPPROTO_TCP:
2089         BUMP_MIB(ill->ill_ip_mib, tcpIfStatsInErrs);
2090
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_rcv_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_rcv_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_rcv_attr_t *ira)
2193 {
2194     ill_t             *ill = ira->ira_rill;
2195     uint16_t          hck_flags;
2196     uint32_t          cksum;
2197     mblk_t            *mpl;
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 rp_ptr ((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_rcv_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_rcv_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 queue 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_queue_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_rcv)(connp, mp, NULL, ira);
2525         CONN_DEC_REF(connp);
2526         ira->ira_ill = ill;
2527         ira->ira_rill = rill;
2528         return;
2529     }
2530
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;
2539
2540             attrmp = ip_rcv_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_rcv,
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 queue.
2557                  */
2558             }
2559         } else {
2560             SQUEUE_ENTER_ONE(connp->conn_sqp, mp,
2561                 connp->conn_rcv, connp, ira, SQ_FILL,
2562                 SQTAG_IP_TCP_INPUT);
2563         }
2564     } else {
2565         SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_rcv,
2566             connp, ira, ip_queue_flag, SQTAG_IP_TCP_INPUT);
2567     }
2568     ira->ira_ill = ill;
2569     ira->ira_rill = rill;

```

```

2570         return;
2571     }
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;
2577
2578         /* For SCTP, discard broadcast and multicast packets. */
2579         if (iraflags & IRAF_MULTIBROADCAST)
2580             goto discard;
2581
2582         /*
2583          * Since there is no SCTP h/w cksum support yet, just
2584          * clear the flag.
2585          */
2586         DB_CKSUMFLAGS(mp) = 0;
2587
2588         /* Length ensured above */
2589         ASSERT(MBLKL(mp) >= ip_hdr_length + SCTP_COMMON_HDR_LENGTH);
2590         sctph = (sctp_hdr_t *) (rptr + ip_hdr_length);
2591
2592         /* get the ports */
2593         ports = *(uint32_t *) &sctph->sh_sport;
2594
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         }
2624
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     }
2631
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_cntrlid :
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_rcv)(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     /* Checked above */
2730     ASSERT(len >= ip_hdr_length + DCCP_MIN_HEADER_LENGTH);
2732     /* Queue hint */
2733     if (ira->ira_sqp == NULL) {
2734         ira->ira_sqp = ip_queue_get(ira->ira_ring);
2735     }
2737     connp = ipcl_classify_v4(mp, IPPROTO_DCCP, ip_hdr_length,
2738         ira, ipst);
2739     if (connp == NULL) {
2740         cmn_err(CE_NOTE, "ip_input.c: ip_fanout_v4 connp not fou
2741         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2742         dccp_xmit_listeners_reset(mp, ira, ipst, NULL);
2743         return;
2744     }
2746     if (connp->conn_incoming_ifindex != 0 &&
2747         connp->conn_incoming_ifindex != ira->ira_ruifindex) {
2748         cmn_err(CE_NOTE, "ip_input.c: ip_fanout_v4 ifindex probl
2749         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2750         dccp_xmit_listeners_reset(mp, ira, ipst, NULL);
2751         return;
2752     }
2754     if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2755         (iraflags & IRAF_IPSEC_SECURE)) {
2756         mp = ipsec_check_inbound_policy(mp, connp,
2757             ipha, NULL, ira);
2758         if (mp == NULL) {
2759             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2760             ip_drop_input("ipIfStatsInDiscards", mp, ill);
2761             CONN_DEC_REF(connp);
2762             return;
2763         }
2764     }
2766     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2767     ira->ira_ill = ira->ira_rill = NULL;

```

```

2769         SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_rcv,
2770         connp, ira, ip_squeue_flag, SQTAG_IP_DCCP_INPUT);

2772         ira->ira_ill = ill;
2773         ira->ira_rill = rill;
2774         return;
2775 #endif /* ! codereview */
2776     default:
2777         break;
2778     }

2780     /*
2781     * Clear hardware checksumming flag as it is currently only
2782     * used by TCP and UDP.
2783     */
2784     DB_CKSUMFLAGS(mp) = 0;

2786     switch (protocol) {
2787     case IPPROTO_ICMP:
2788         /*
2789         * We need to accomodate icmp messages coming in clear
2790         * until we get everything secure from the wire. If
2791         * icmp_accept_clear_messages is zero we check with
2792         * the global policy and act accordingly. If it is
2793         * non-zero, we accept the message without any checks.
2794         * But *this does not mean* that this will be delivered
2795         * to RAW socket clients. By accepting we might send
2796         * replies back, change our MTU value etc.,
2797         * but delivery to the ULP/clients depends on their
2798         * policy dispositions.
2799         */
2800         if (ipst->ips_icmp_accept_clear_messages == 0) {
2801             mp = ipsec_check_global_policy(mp, NULL,
2802             ipha, NULL, ira, ns);
2803             if (mp == NULL)
2804                 return;
2805         }

2807         /*
2808         * On a labeled system, we have to check whether the zone
2809         * itself is permitted to receive raw traffic.
2810         */
2811         if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
2812             if (!tsol_can_accept_raw(mp, ira, B_FALSE)) {
2813                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
2814                 ip_drop_input("tsol_can_accept_raw", mp, ill);
2815                 freemsg(mp);
2816                 return;
2817             }
2818         }

2820         /*
2821         * ICMP header checksum, including checksum field,
2822         * should be zero.
2823         */
2824         if (IP_CSUM(mp, ip_hdr_length, 0)) {
2825             BUMP_MIB(&ipst->ips_icmp_mib, icmpInCksumErrs);
2826             ip_drop_input("icmpInCksumErrs", mp, ill);
2827             freemsg(mp);
2828             return;
2829         }
2830         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2831         mp = icmp_inbound_v4(mp, ira);
2832         if (mp == NULL) {
2833             /* No need to pass to RAW sockets */

```

```

2834         return;
2835     }
2836     break;

2838     case IPPROTO_IGMP:
2839         /*
2840         * If we are not willing to accept IGMP packets in clear,
2841         * then check with global policy.
2842         */
2843         if (ipst->ips_igmp_accept_clear_messages == 0) {
2844             mp = ipsec_check_global_policy(mp, NULL,
2845             ipha, NULL, ira, ns);
2846             if (mp == NULL)
2847                 return;
2848         }
2849         if ((ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2850             !tsol_can_accept_raw(mp, ira, B_TRUE)) {
2851             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2852             ip_drop_input("ipIfStatsInDiscards", mp, ill);
2853             freemsg(mp);
2854             return;
2855         }
2856         /*
2857         * Validate checksum
2858         */
2859         if (IP_CSUM(mp, ip_hdr_length, 0)) {
2860             ++ipst->ips_igmpstat.igps_rcv_badsum;
2861             ip_drop_input("igps_rcv_badsum", mp, ill);
2862             freemsg(mp);
2863             return;
2864         }

2866         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2867         mp = igmp_input(mp, ira);
2868         if (mp == NULL) {
2869             /* Bad packet - discarded by igmp_input */
2870             return;
2871         }
2872         break;
2873     case IPPROTO_PIM:
2874         /*
2875         * If we are not willing to accept PIM packets in clear,
2876         * then check with global policy.
2877         */
2878         if (ipst->ips_pim_accept_clear_messages == 0) {
2879             mp = ipsec_check_global_policy(mp, NULL,
2880             ipha, NULL, ira, ns);
2881             if (mp == NULL)
2882                 return;
2883         }
2884         if ((ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2885             !tsol_can_accept_raw(mp, ira, B_TRUE)) {
2886             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2887             ip_drop_input("ipIfStatsInDiscards", mp, ill);
2888             freemsg(mp);
2889             return;
2890         }
2891         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);

2893         /* Checksum is verified in pim_input */
2894         mp = pim_input(mp, ira);
2895         if (mp == NULL) {
2896             /* Bad packet - discarded by pim_input */
2897             return;
2898         }
2899         break;

```

```

2900     case IPPROTO_AH:
2901     case IPPROTO_ESP: {
2902         /*
2903          * Fast path for AH/ESP.
2904          */
2905         netstack_t *ns = ipst->ips_netstack;
2906         ipsec_stack_t *ipss = ns->netstack_ipsec;
2907
2908         IP_STAT(ipst, ipsec_proto_ahesp);
2909
2910         if (!ipsec_loaded(ipss)) {
2911             ip_proto_not_sup(mp, ira);
2912             return;
2913         }
2914
2915         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2916         /* select inbound SA and have IPsec process the pkt */
2917         if (protocol == IPPROTO_ESP) {
2918             esph_t *esph;
2919             boolean_t esp_in_udp_sa;
2920             boolean_t esp_in_udp_packet;
2921
2922             mp = ipsec_inbound_esp_sa(mp, ira, &esph);
2923             if (mp == NULL)
2924                 return;
2925
2926             ASSERT(esph != NULL);
2927             ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
2928             ASSERT(ira->ira_ipsec_esp_sa != NULL);
2929             ASSERT(ira->ira_ipsec_esp_sa->ipsa_input_func != NULL);
2930
2931             esp_in_udp_sa = ((ira->ira_ipsec_esp_sa->ipsa_flags &
2932             IPSA_F_NATT) != 0);
2933             esp_in_udp_packet =
2934             (ira->ira_flags & IRAF_ESP_UDP_PORTS) != 0;
2935
2936             /*
2937              * The following is a fancy, but quick, way of saying:
2938              * ESP-in-UDP SA and Raw ESP packet --> drop
2939              * OR
2940              * ESP SA and ESP-in-UDP packet --> drop
2941              */
2942             if (esp_in_udp_sa != esp_in_udp_packet) {
2943                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2944                 ip_drop_packet(mp, B_TRUE, ira->ira_ill,
2945                 DROPPER(ipss, ipds_esp_no_sa),
2946                 &ipss->ipsec_dropper);
2947                 return;
2948             }
2949             mp = ira->ira_ipsec_esp_sa->ipsa_input_func(mp, esph,
2950             ira);
2951         } else {
2952             ah_t *ah;
2953
2954             mp = ipsec_inbound_ah_sa(mp, ira, &ah);
2955             if (mp == NULL)
2956                 return;
2957
2958             ASSERT(ah != NULL);
2959             ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
2960             ASSERT(ira->ira_ipsec_ah_sa != NULL);
2961             ASSERT(ira->ira_ipsec_ah_sa->ipsa_input_func != NULL);
2962             mp = ira->ira_ipsec_ah_sa->ipsa_input_func(mp, ah,
2963             ira);
2964         }

```

```

2966         if (mp == NULL) {
2967             /*
2968              * Either it failed or is pending. In the former case
2969              * ipIfStatsInDiscards was increased.
2970              */
2971             return;
2972         }
2973         /* we're done with IPsec processing, send it up */
2974         ip_input_post_ipsec(mp, ira);
2975         return;
2976     }
2977     case IPPROTO_ENCAP: {
2978         ipha_t         *inner_ipha;
2979
2980         /*
2981          * Handle self-encapsulated packets (IP-in-IP where
2982          * the inner addresses == the outer addresses).
2983          */
2984         if ((uchar_t *)ipha + ip_hdr_length + sizeof (ipha_t) >
2985             mp->b_wptr) {
2986             if (ira->ira_pktlen <
2987                 ip_hdr_length + sizeof (ipha_t)) {
2988                 BUMP_MIB(ill->ill_ip_mib,
2989                 ipIfStatsInTruncatedPkts);
2990                 ip_drop_input("ipIfStatsInTruncatedPkts",
2991                 mp, ill);
2992                 freemsg(mp);
2993                 return;
2994             }
2995             ipha = ip_pullup(mp, (uchar_t *)ipha + ip_hdr_length +
2996             sizeof (ipha_t) - mp->b_rptr, ira);
2997             if (ipha == NULL) {
2998                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2999                 ip_drop_input("ipIfStatsInDiscards", mp, ill);
3000                 freemsg(mp);
3001                 return;
3002             }
3003             inner_ipha = (ipha_t *)((uchar_t *)ipha + ip_hdr_length);
3004             /*
3005              * Check the sanity of the inner IP header.
3006              */
3007             if ((IPH_HDR_VERSION(inner_ipha) != IPV4_VERSION)) {
3008                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3009                 ip_drop_input("ipIfStatsInDiscards", mp, ill);
3010                 freemsg(mp);
3011                 return;
3012             }
3013             if (IPH_HDR_LENGTH(inner_ipha) < sizeof (ipha_t)) {
3014                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3015                 ip_drop_input("ipIfStatsInDiscards", mp, ill);
3016                 freemsg(mp);
3017                 return;
3018             }
3019             if (inner_ipha->ipha_src != ipha->ipha_src ||
3020                 inner_ipha->ipha_dst != ipha->ipha_dst) {
3021                 /* We fallthru to iptun fanout below */
3022                 goto iptun;
3023             }
3024         }
3025
3026         /*
3027          * Self-encapsulated tunnel packet. Remove
3028          * the outer IP header and fanout again.
3029          * We also need to make sure that the inner
3030          * header is pulled up until options.
3031          */

```

```

3032     mp->b_rptr = (uchar_t *)inner_ipha;
3033     ipha = inner_ipha;
3034     ip_hdr_length = IPH_HDR_LENGTH(ipha);
3035     if ((uchar_t *)ipha + ip_hdr_length > mp->b_wptr) {
3036         if (ira->ira_pktlen <
3037             (uchar_t *)ipha + ip_hdr_length - mp->b_rptr) {
3038             BUMP_MIB(ill->ill_ip_mib,
3039                 ipIfStatsInTruncatedPkts);
3040             ip_drop_input("ipIfStatsInTruncatedPkts",
3041                 mp, ill);
3042             freemsg(mp);
3043             return;
3044         }
3045         ipha = ip_pullup(mp,
3046             (uchar_t *)ipha + ip_hdr_length - mp->b_rptr, ira);
3047         if (ipha == NULL) {
3048             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3049             ip_drop_input("ipIfStatsInDiscards", mp, ill);
3050             freemsg(mp);
3051             return;
3052         }
3053     }
3054     if (ip_hdr_length > sizeof(ipha_t)) {
3055         /* We got options on the inner packet. */
3056         ipaddr_t     dst = ipha->ipha_dst;
3057         int           error = 0;

3059         dst = ip_input_options(ipha, dst, mp, ira, &error);
3060         if (error != 0) {
3061             /*
3062              * An ICMP error has been sent and the packet
3063              * has been dropped.
3064              */
3065             return;
3066         }
3067         if (dst != ipha->ipha_dst) {
3068             /*
3069              * Someone put a source-route in
3070              * the inside header of a self-
3071              * encapsulated packet. Drop it
3072              * with extreme prejudice and let
3073              * the sender know.
3074              */
3075             ip_drop_input("ICMP_SOURCE_ROUTE_FAILED",
3076                 mp, ill);
3077             icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED,
3078                 ira);
3079             return;
3080         }
3081     }
3082     if (!(ira->ira_flags & IRAF_IPSEC_SECURE)) {
3083         /*
3084          * This means that somebody is sending
3085          * Self-encapsualted packets without AH/ESP.
3086          *
3087          * Send this packet to find a tunnel endpoint.
3088          * if I can't find one, an ICMP
3089          * PROTOCOL_UNREACHABLE will get sent.
3090          */
3091         protocol = ipha->ipha_protocol;
3092         ira->ira_protocol = protocol;
3093         goto iptun;
3094     }

3096     /* Update based on removed IP header */
3097     ira->ira_ip_hdr_length = ip_hdr_length;

```

```

3098     ira->ira_pktlen = ntohs(ipha->ipha_length);
3099
3100     if (ira->ira_flags & IRAF_IPSEC_DECAPS) {
3101         /*
3102          * This packet is self-encapsulated multiple
3103          * times. We don't want to recurse infinitely.
3104          * To keep it simple, drop the packet.
3105          */
3106         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3107         ip_drop_input("ipIfStatsInDiscards", mp, ill);
3108         freemsg(mp);
3109         return;
3110     }
3111     ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
3112     ira->ira_flags |= IRAF_IPSEC_DECAPS;

3114     ip_input_post_ipsec(mp, ira);
3115     return;
3116 }

3118 iptun: /* IPPROTO_ENCAPS that is not self-encapsulated */
3119 case IPPROTO_IPV6:
3120     /* iptun will verify trusted label */
3121     connp = ipcl_classify_v4(mp, protocol, ip_hdr_length,
3122         ira, ipst);
3123     if (connp != NULL) {
3124         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
3125         ira->ira_ill = ira->ira_rill = NULL;
3126         (connp->conn_rcv)(connp, mp, NULL, ira);
3127         CONN_DEC_REF(connp);
3128         ira->ira_ill = ill;
3129         ira->ira_rill = rill;
3130         return;
3131     }
3132     /* FALLTHRU */
3133 default:
3134     /*
3135      * On a labeled system, we have to check whether the zone
3136      * itself is permitted to receive raw traffic.
3137      */
3138     if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
3139         if (!tsol_can_accept_raw(mp, ira, B_FALSE)) {
3140             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3141             ip_drop_input("ipIfStatsInDiscards", mp, ill);
3142             freemsg(mp);
3143             return;
3144         }
3145     }
3146     break;
3147 }

3149 /*
3150  * The above input functions may have returned the pulled up message.
3151  * So ipha need to be reinitialized.
3152  */
3153 ipha = (ipha_t *)mp->b_rptr;
3154 ira->ira_protocol = protocol = ipha->ipha_protocol;
3155 if (ipst->ips_ipcl_proto_fanout_v4[protocol].connf_head == NULL) {
3156     /*
3157      * No user-level listener for these packets packets.
3158      * Check for IPPROTO_ENCAP...
3159      */
3160     if (protocol == IPPROTO_ENCAP && ipst->ips_ip_g_router) {
3161         /*
3162          * Check policy here,
3163          * THEN ship off to ip_mroute_decap().

```

```
3164      *
3165      * BTW, If I match a configured IP-in-IP
3166      * tunnel above, this path will not be reached, and
3167      * ip_mroute_decap will never be called.
3168      */
3169      mp = ipsec_check_global_policy(mp, connp,
3170      ipha, NULL, ira, ns);
3171      if (mp != NULL) {
3172          ip_mroute_decap(mp, ira);
3173      } /* Else we already freed everything! */
3174      } else {
3175          ip_proto_not_sup(mp, ira);
3176      }
3177      return;
3178  }
3179
3180  /*
3181  * Handle fanout to raw sockets. There
3182  * can be more than one stream bound to a particular
3183  * protocol. When this is the case, each one gets a copy
3184  * of any incoming packets.
3185  */
3186  ASSERT(ira->ira_protocol == ipha->ipha_protocol);
3187  ip_fanout_proto_v4(mp, ipha, ira);
3188  return;
3189
3190 discard:
3191  BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3192  ip_drop_input("ipIfStatsInDiscards", mp, ill);
3193  freemsg(mp);
3194 #undef rptra
3195 }
```

new/usr/src/uts/common/inet/ip/ip_output.c

1

```
*****
74252 Wed Jun 13 12:05:01 2012
new/usr/src/uts/common/inet/ip/ip_output.c
%B
*****
_____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 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
```

new/usr/src/uts/common/inet/ip/ip_output.c

2

```
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 iaflags, 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 ((iaflags & 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     cksum = 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 }
1755
1756 /* ULP puts the checksum field in the first mblk */
1757 ASSERT(((uchar_t *) cksum) + sizeof (uint16_t) <= mp->b_wptr);
1758
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;
1766
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     *cksum = 0;
1775     DB_CKSUMFLAGS(mp) |= HCK_FULLCKSUM;
1776
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 += *(cksum);
1802     cksum = (cksum & 0xFFFF) + (cksum >> 16);
1803     *(cksum) = (cksum & 0xFFFF) + (cksum >> 16);
1804
1805     /*
1806      * Offsets are relative to beginning of IP header.
1807      */
1808     DB_CKSUMSTART(mp) = ip_hdr_length;
1809     DB_CKSUMSTUFF(mp) = (uint8_t *) cksum - (uint8_t *) ipha;
1810     DB_CKSUMEND(mp) = pktlen;
1811     DB_CKSUMFLAGS(mp) |= HCK_PARTIALCKSUM;
1812
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 }
1824
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     iaflags = ixa->ixa_flags;
1842     ill_t         *ill;
1843
1844     ASSERT(ixa->ixa_nce != NULL);
1845     ill = ixa->ixa_nce->nce_ill;
1846
1847     if (iaflags & IXAF_DONTROUTE)
1848         ipha->ipha_ttl = 1;
1849
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_rcv_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_rcv_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 queue */
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_LABELLED))
2122         ira_cleanup(&iras, B_FALSE);
2123  }

2125 /*
2126  * Post fragmentation function for IRE_MULTICAST and IRE_BROADCAST which

```

```

2127 * looks at the IXAF_LOOPBACK_COPY flag.
2128 * Common for IPv4 and IPv6.
2129 *
2130 * If the loopback copy fails (due to no memory) but we send the packet out
2131 * on the wire we return no failure. Only in the case we suppress the wire
2132 * sending do we take the loopback failure into account.
2133 *
2134 * Note that we do not perform DTRACE_IP7 and FW_HOOKS for the looped back copy.
2135 * Those operations are performed on this packet in ip_xmit() and it would
2136 * be odd to do it twice for the same packet.
2137 */
2138 int
2139 ip_postfrag_loopcheck(mblk_t *mp, nce_t *nce, iaflags_t iaflags,
2140 uint_t pkt_len, uint32_t xmit_hint, zoneid_t szone, zoneid_t nolzid,
2141 uintptr_t *ixacookie)
2142 {
2143     ill_t      *ill = nce->nice_ill;
2144     int
2145     error = 0;
2146
2147     /*
2148     * Check for IXAF_LOOPBACK_COPY - send a copy to ip as if the driver
2149     * had looped it back
2150     */
2151     if (iaflags & IXAF_LOOPBACK_COPY) {
2152         mblk_t      *mpl;
2153
2154         mpl = copymsg(mp);
2155         if (mpl == NULL) {
2156             /* Failed to deliver the loopback copy. */
2157             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2158             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
2159             error = ENOBUFS;
2160         } else {
2161             ip_postfrag_loopback(mpl, nce, iaflags, pkt_len,
2162                                 nolzid);
2163         }
2164     }
2165
2166     /*
2167     * If TTL = 0 then only do the loopback to this host i.e. we are
2168     * done. We are also done if this was the
2169     * loopback interface since it is sufficient
2170     * to loopback one copy of a multicast packet.
2171     */
2172     if (iaflags & IXAF_IS_IPV4) {
2173         ipha_t *ipha = (ipha_t *)mp->b_rptr;
2174
2175         if (ipha->ipha_ttl == 0) {
2176             ip_drop_output("multicast ipha_ttl not sent to wire",
2177                           mp, ill);
2178             freemsg(mp);
2179             return (error);
2180         }
2181     } else {
2182         ip6_t      *ip6h = (ip6_t *)mp->b_rptr;
2183
2184         if (ip6h->ip6_hops == 0) {
2185             ip_drop_output("multicast ipha_ttl not sent to wire",
2186                           mp, ill);
2187             freemsg(mp);
2188             return (error);
2189         }
2190     }
2191     if (nce->nice_ill->ill_wq == NULL) {
2192         /* Loopback interface */
2193         ip_drop_output("multicast on lo0 not sent to wire", mp, ill);

```

```

2193         freemsg(mp);
2194         return (error);
2195     }
2196
2197     return (ip_xmit(mp, nce, iaflags, pkt_len, xmit_hint, szone, 0,
2198                    ixacookie));
2199 }
2200
2201 /*
2202 * Post fragmentation function for RTF_MULTIRT routes.
2203 * Since IRE_BROADCASTs can have RTF_MULTIRT, this function
2204 * checks IXAF_LOOPBACK_COPY.
2205 *
2206 * If no packet is sent due to failures then we return an errno, but if at
2207 * least one succeeded we return zero.
2208 */
2209 int
2210 ip_postfrag_multirt_v4(mblk_t *mp, nce_t *nce, iaflags_t iaflags,
2211 uint_t pkt_len, uint32_t xmit_hint, zoneid_t szone, zoneid_t nolzid,
2212 uintptr_t *ixacookie)
2213 {
2214     irb_t      *irb;
2215     ipha_t      *ipha = (ipha_t *)mp->b_rptr;
2216     ire_t      *ire;
2217     ire_t      *irel;
2218     mblk_t      *mpl;
2219     nce_t      *nce;
2220     ill_t      *ill = nce->nice_ill;
2221     ill_t      *illl;
2222     ip_stack_t *ipst = ill->ill_ipst;
2223     int
2224     error = 0;
2225     int
2226     num_sent = 0;
2227     int
2228     err;
2229     uint_t      ire_type;
2230     ipaddr_t    nexthop;
2231
2232     ASSERT(iaflags & IXAF_IS_IPV4);
2233
2234     /* Check for IXAF_LOOPBACK_COPY */
2235     if (iaflags & IXAF_LOOPBACK_COPY) {
2236         mblk_t *mpl;
2237
2238         mpl = copymsg(mp);
2239         if (mpl == NULL) {
2240             /* Failed to deliver the loopback copy. */
2241             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2242             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
2243             error = ENOBUFS;
2244         } else {
2245             ip_postfrag_loopback(mpl, nce, iaflags, pkt_len,
2246                                 nolzid);
2247         }
2248     }
2249
2250     /*
2251     * Loop over RTF_MULTIRT for ipha_dst in the same bucket. Send
2252     * a copy to each one.
2253     * Use the nce (nexthop) and ipha_dst to find the ire.
2254     * MULTIRT is not designed to work with shared-IP zones thus we don't
2255     * need to pass a zoneid or a label to the IRE lookup.
2256     */
2257     if (V4_PART_OF_V6(nce->nice_addr) == ipha->ipha_dst) {
2258         /* Broadcast and multicast case */
2259         ire = ire_fhtable_lookup_v4(ipha->ipha_dst, 0, 0, 0,
2260                                    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_fhtable_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 (irel = irb->irb_ire; irel != NULL; irel = irel->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(irel) ||
2288             !(irel->ire_flags & RTF_MULTIRT) ||
2289             irel->ire_type != ire->ire_type)
2290             continue;

2292         /* Do the ire argument one after the loop */
2293         if (irel == ire)
2294             continue;

2296         ill1 = ire_nexthop_ill(irel);
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 irel.
2304              */
2305             ire_t *ire2;
2306             uint_t match_flags = MATCH_IRE_DSTONLY;

2308             if (irel->ire_ill != NULL)
2309                 match_flags |= MATCH_IRE_ILL;
2310             ire2 = ire_route_recursive_impl_v4(irel,
2311                 irel->ire_addr, irel->ire_type, irel->ire_ill,
2312                 irel->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(irel);
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 = irel->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 = irel->ire_type; /* Doesn't matter */
2336         nexthop = irel->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     ncel = 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(ncel);
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     irel->ire_ob_pkt_count++;
2375     err = ip_xmit(mp1, ncel, ixaflags, pkt_len, xmit_hint, szone,
2376         0, ixacookie);
2377     if (err == 0)
2378         num_sent++;
2379     else
2380         error = err;
2381     nce_refrele(ncel);
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 = ix->ixa_ire;

2410     if (!(ire->ire_type & (IRE_LOCAL | IRE_LOOPBACK)))
2411         return (B_FALSE);

2413     return (ixa->ixa_ire->ire_generation == ix->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 = ix->ixa_ire->ire_ill;
2427     ipha_t        *ipha = NULL;
2428     ip6_t         *ip6h = NULL;
2429     ip_stack_t    *ipst = ix->ixa_ipst;
2430     iaflags_t     iaflags = ix->ixa_flags;
2431     ip_rcv_attr_t iras;
2432     int           error;

2434     ASSERT(mp != NULL);

2436     if (iaflags & 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 = ix->ixa_ire->ire_zoneid;
2454                 szone = ix->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 policies */
2494 if (peer_connp != NULL) {
2495     /* Map ix to ira including IPsec policies. */
2496     ipsec_out_to_in(ix, 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 = ix->ixa_ire->ire_zoneid;
2518             szone = ix->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 policies */
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 }
```

```

*****
84408 Wed Jun 13 12:05:02 2012
new/usr/src/uts/common/inet/ip/ipclassifier.c
%B
*****
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 */
21 /*
22 * Copyright (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
23 */

25 /*
26 * IP PACKET CLASSIFIER
27 *
28 * The IP packet classifier provides mapping between IP packets and persistent
29 * connection state for connection-oriented protocols. It also provides
30 * interface for managing connection states.
31 *
32 * The connection state is kept in conn_t data structure and contains, among
33 * other things:
34 *
35 *     o local/remote address and ports
36 *     o Transport protocol
37 *     o queue for the connection (for TCP only)
38 *     o reference counter
39 *     o Connection state
40 *     o hash table linkage
41 *     o interface/ire information
42 *     o credentials
43 *     o ipsec policy
44 *     o send and receive functions.
45 *     o mutex lock.
46 *
47 * Connections use a reference counting scheme. They are freed when the
48 * reference counter drops to zero. A reference is incremented when connection
49 * is placed in a list or table, when incoming packet for the connection arrives
50 * and when connection is processed via queue (queue processing may be
51 * asynchronous and the reference protects the connection from being destroyed
52 * before its processing is finished).
53 *
54 * conn_rcv is used to pass up packets to the ULP.
55 * For TCP conn_rcv changes. It is tcp_input_listener_unbound initially for
56 * a listener, and changes to tcp_input_listener as the listener has picked a
57 * good queue. For other cases it is set to tcp_input_data.
58 *
59 * conn_rcvicmp is used to pass up ICMP errors to the ULP.
60 *
61 * Classifier uses several hash tables:

```

```

62 *
63 *     ipcl_conn_fanout:     contains all TCP connections in CONNECTED state
64 *     ipcl_bind_fanout:    contains all connections in BOUND state
65 *     ipcl_proto_fanout:   IPv4 protocol fanout
66 *     ipcl_proto_fanout_v6: IPv6 protocol fanout
67 *     ipcl_udp_fanout:     contains all UDP connections
68 *     ipcl_iptun_fanout:   contains all IP tunnel connections
69 *     ipcl_globalhash_fanout: contains all connections
70 *
71 * The ipcl_globalhash_fanout is used for any walkers (like snmp and Clustering)
72 * which need to view all existing connections.
73 *
74 * All tables are protected by per-bucket locks. When both per-bucket lock and
75 * connection lock need to be held, the per-bucket lock should be acquired
76 * first, followed by the connection lock.
77 *
78 * All functions doing search in one of these tables increment a reference
79 * counter on the connection found (if any). This reference should be dropped
80 * when the caller has finished processing the connection.
81 *
82 *
83 * INTERFACES:
84 * =====
85 *
86 * Connection Lookup:
87 * -----
88 *
89 * conn_t *ipcl_classify_v4(mp, protocol, hdr_len, ira, ip_stack)
90 * conn_t *ipcl_classify_v6(mp, protocol, hdr_len, ira, ip_stack)
91 *
92 * Finds connection for an incoming IPv4 or IPv6 packet. Returns NULL if
93 * it can't find any associated connection. If the connection is found, its
94 * reference counter is incremented.
95 *
96 *     mp:     mblock, containing packet header. The full header should fit
97 *             into a single mblock. It should also contain at least full IP
98 *             and TCP or UDP header.
99 *
100 *     protocol: Either IPPROTO_TCP or IPPROTO_UDP.
101 *
102 *     hdr_len: The size of IP header. It is used to find TCP or UDP header in
103 *              the packet.
104 *
105 *     ira->ira_zoneid: The zone in which the returned connection must be; the
106 *                      zoneid corresponding to the ire_zoneid on the IRE located for
107 *                      the packet's destination address.
108 *
109 *     ira->ira_flags: Contains the IRAF_TX_MAC_EXEMPTABLE and
110 *                    IRAF_TX_SHARED_ADDR flags
111 *
112 * For TCP connections, the lookup order is as follows:
113 *     5-tuple {src, dst, protocol, local port, remote port}
114 *     lookup in ipcl_conn_fanout table.
115 *     3-tuple {dst, remote port, protocol} lookup in
116 *     ipcl_bind_fanout table.
117 *
118 * For UDP connections, a 5-tuple {src, dst, protocol, local port,
119 * remote port} lookup is done on ipcl_udp_fanout. Note that,
120 * these interfaces do not handle cases where a packets belongs
121 * to multiple UDP clients, which is handled in IP itself.
122 *
123 * If the destination IRE is ALL_ZONES (indicated by zoneid), then we must
124 * determine which actual zone gets the segment. This is used only in a
125 * labeled environment. The matching rules are:
126 *
127 *     - 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 *, tcpa_t *, int, ip_stack);
158 * conn_t *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpa_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 * Arguments :
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 *     Arguments :
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 */
235 *
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>
246 *
247 #include <sys/system.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>
254 *
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>

```



```

260 #include <inet/ip_impl.h>
261 #include <inet/udp_impl.h>
262 #include <inet/dccp/dccp_impl.h>
263 #endif /* ! codereview */
264 #include <inet/sctp_ip.h>
265 #include <inet/sctp/sctp_impl.h>
266 #include <inet/rawip_impl.h>
267 #include <inet/rts_impl.h>
268 #include <inet/iptun/iptun_impl.h>

270 #include <sys/cpuvar.h>

272 #include <inet/ipclassifier.h>
273 #include <inet/tcp.h>
274 #include <inet/ipsec_impl.h>

276 #include <sys/tsol/tnet.h>
277 #include <sys/sockio.h>

279 /* Old value for compatibility. Setable in /etc/system */
280 uint_t tcp_conn_hash_size = 0;

282 /* New value. Zero means choose automatically. Setable in /etc/system */
283 uint_t ipcl_conn_hash_size = 0;
284 uint_t ipcl_conn_hash_memfactor = 8192;
285 uint_t ipcl_conn_hash_maxsize = 82500;

287 /* bind/dccp/udp fanout table size */
288 /* bind/udp fanout table size */
288 uint_t ipcl_bind_fanout_size = 512;
289 uint_t ipcl_dccp_fanout_size = 512;
290 #endif /* ! codereview */
291 uint_t ipcl_udp_fanout_size = 16384;

293 /* Raw socket fanout size. Must be a power of 2. */
294 uint_t ipcl_raw_fanout_size = 256;

296 /*
297 * The IPCL_IPTUN_HASH() function works best with a prime table size. We
298 * expect that most large deployments would have hundreds of tunnels, and
299 * thousands in the extreme case.
300 */
301 uint_t ipcl_iptun_fanout_size = 6143;

303 /*
304 * Power of 2^N Primes useful for hashing for N of 0-28,
305 * these primes are the nearest prime <= 2^N - 2^(N-2).
306 */

308 #define P2Ps() {0, 0, 0, 5, 11, 23, 47, 89, 191, 383, 761, 1531, 3067, \
309             6143, 12281, 24571, 49139, 98299, 196597, 393209, \
310             786431, 1572853, 3145721, 6291449, 12582893, 25165813, \
311             50331599, 100663291, 201326557, 0}

313 /*
314 * wrapper structure to ensure that conn and what follows it (tcp_t, etc)
315 * are aligned on cache lines.
316 */
317 typedef union itc_s {
318     conn_t itc_conn;
319     char itcu_filler[CACHE_ALIGN(conn_s)];
320 } itc_t;

322 struct kmem_cache *tcp_conn_cache;
323 struct kmem_cache *ip_conn_cache;
324 extern struct kmem_cache *sctp_conn_cache;

```

```

325 struct kmem_cache *udp_conn_cache;
326 struct kmem_cache *rawip_conn_cache;
327 struct kmem_cache *rts_conn_cache;
328 struct kmem_cache *dccp_conn_cache;
329 #endif /* ! codereview */

331 extern void tcp_timermp_free(tcp_t *);
332 extern mblk_t *tcp_timermp_alloc(int);

334 static int ip_conn_constructor(void *, void *, int);
335 static void ip_conn_destructor(void *, void *);

337 static int tcp_conn_constructor(void *, void *, int);
338 static void tcp_conn_destructor(void *, void *);

340 static int udp_conn_constructor(void *, void *, int);
341 static void udp_conn_destructor(void *, void *);

343 static int rawip_conn_constructor(void *, void *, int);
344 static void rawip_conn_destructor(void *, void *);

346 static int rts_conn_constructor(void *, void *, int);
347 static void rts_conn_destructor(void *, void *);

349 static int dccp_conn_constructor(void *, void *, int);
350 static void dccp_conn_destructor(void *, void *);

352 #endif /* ! codereview */
353 /*
354 * Global (for all stack instances) init routine
355 */
356 void
357 ipcl_g_init(void)
358 {
359     ip_conn_cache = kmem_cache_create("ip_conn_cache",
360                                     sizeof(conn_t), CACHE_ALIGN_SIZE,
361                                     ip_conn_constructor, ip_conn_destructor,
362                                     NULL, NULL, NULL, 0);

364     tcp_conn_cache = kmem_cache_create("tcp_conn_cache",
365                                       sizeof(itc_t) + sizeof(tcp_t), CACHE_ALIGN_SIZE,
366                                       tcp_conn_constructor, tcp_conn_destructor,
367                                       tcp_conn_reclaim, NULL, NULL, 0);

369     udp_conn_cache = kmem_cache_create("udp_conn_cache",
370                                       sizeof(itc_t) + sizeof(udp_t), CACHE_ALIGN_SIZE,
371                                       udp_conn_constructor, udp_conn_destructor,
372                                       NULL, NULL, NULL, 0);

374     rawip_conn_cache = kmem_cache_create("rawip_conn_cache",
375                                         sizeof(itc_t) + sizeof(icmp_t), CACHE_ALIGN_SIZE,
376                                         rawip_conn_constructor, rawip_conn_destructor,
377                                         NULL, NULL, NULL, 0);

379     rts_conn_cache = kmem_cache_create("rts_conn_cache",
380                                       sizeof(itc_t) + sizeof(rts_t), CACHE_ALIGN_SIZE,
381                                       rts_conn_constructor, rts_conn_destructor,
382                                       NULL, NULL, NULL, 0);

384     /* XXX:DCCP reclaim */
385     dccp_conn_cache = kmem_cache_create("dccp_conn_cache",
386                                       sizeof(itc_t) + sizeof(dccp_t), CACHE_ALIGN_SIZE,
387                                       dccp_conn_constructor, dccp_conn_destructor,
388                                       NULL, NULL, NULL, 0);
389 #endif /* ! codereview */
390 }

```

```

392 /*
393  * ipclassifier initialization routine, sets up hash tables.
394  */
395 void
396 ipcl_init(ip_stack_t *ipst)
397 {
398     int i;
399     int sizes[] = P2Ps();
400
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;
410
411         ipst->ips_ipcl_conn_fanout_size =
412             (freemem * PAGESIZE) / ipcl_conn_hash_memfactor;
413
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     }
419
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     }
429
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;
437
438     ASSERT(ipst->ips_ipcl_conn_fanout == NULL);
439
440     ipst->ips_ipcl_conn_fanout = kmem_zalloc(
441         ipst->ips_ipcl_conn_fanout_size * sizeof (connf_t), KM_SLEEP);
442
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     }
447
448     ipst->ips_ipcl_bind_fanout = kmem_zalloc(
449         ipst->ips_ipcl_bind_fanout_size * sizeof (connf_t), KM_SLEEP);
450
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     }
455
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     }
462
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     }
469
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);
473
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     }
480
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     }
487
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     }
494
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     }
501
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 }
510
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 }

```

```

523 /*
524  * All user-level and kernel use of the stack must be gone
525  * by now.
526  */
527 void
528 ipcl_destroy(ip_stack_t *ipst)
529 {
530     int i;

532     for (i = 0; i < ipst->ips_ipcl_conn_fanout_size; i++) {
533         ASSERT(ipst->ips_ipcl_conn_fanout[i].connf_head == NULL);
534         mutex_destroy(&ipst->ips_ipcl_conn_fanout[i].connf_lock);
535     }
536     kmem_free(ipst->ips_ipcl_conn_fanout, ipst->ips_ipcl_conn_fanout_size *
537             sizeof (connf_t));
538     ipst->ips_ipcl_conn_fanout = NULL;

540     for (i = 0; i < ipst->ips_ipcl_bind_fanout_size; i++) {
541         ASSERT(ipst->ips_ipcl_bind_fanout[i].connf_head == NULL);
542         mutex_destroy(&ipst->ips_ipcl_bind_fanout[i].connf_lock);
543     }
544     kmem_free(ipst->ips_ipcl_bind_fanout, ipst->ips_ipcl_bind_fanout_size *
545             sizeof (connf_t));
546     ipst->ips_ipcl_bind_fanout = NULL;

548     for (i = 0; i < IPPROTO_MAX; i++) {
549         ASSERT(ipst->ips_ipcl_proto_fanout_v4[i].connf_head == NULL);
550         mutex_destroy(&ipst->ips_ipcl_proto_fanout_v4[i].connf_lock);
551     }
552     kmem_free(ipst->ips_ipcl_proto_fanout_v4,
553             IPPROTO_MAX * sizeof (connf_t));
554     ipst->ips_ipcl_proto_fanout_v4 = NULL;

556     for (i = 0; i < IPPROTO_MAX; i++) {
557         ASSERT(ipst->ips_ipcl_proto_fanout_v6[i].connf_head == NULL);
558         mutex_destroy(&ipst->ips_ipcl_proto_fanout_v6[i].connf_lock);
559     }
560     kmem_free(ipst->ips_ipcl_proto_fanout_v6,
561             IPPROTO_MAX * sizeof (connf_t));
562     ipst->ips_ipcl_proto_fanout_v6 = NULL;

564     for (i = 0; i < ipst->ips_ipcl_udp_fanout_size; i++) {
565         ASSERT(ipst->ips_ipcl_udp_fanout[i].connf_head == NULL);
566         mutex_destroy(&ipst->ips_ipcl_udp_fanout[i].connf_lock);
567     }
568     kmem_free(ipst->ips_ipcl_udp_fanout, ipst->ips_ipcl_udp_fanout_size *
569             sizeof (connf_t));
570     ipst->ips_ipcl_udp_fanout = NULL;

572     for (i = 0; i < ipst->ips_ipcl iptun_fanout_size; i++) {
573         ASSERT(ipst->ips_ipcl iptun_fanout[i].connf_head == NULL);
574         mutex_destroy(&ipst->ips_ipcl iptun_fanout[i].connf_lock);
575     }
576     kmem_free(ipst->ips_ipcl iptun_fanout,
577             ipst->ips_ipcl iptun_fanout_size * sizeof (connf_t));
578     ipst->ips_ipcl iptun_fanout = NULL;

580     for (i = 0; i < ipst->ips_ipcl_raw_fanout_size; i++) {
581         ASSERT(ipst->ips_ipcl_raw_fanout[i].connf_head == NULL);
582         mutex_destroy(&ipst->ips_ipcl_raw_fanout[i].connf_lock);
583     }
584     kmem_free(ipst->ips_ipcl_raw_fanout, ipst->ips_ipcl_raw_fanout_size *
585             sizeof (connf_t));
586     ipst->ips_ipcl_raw_fanout = NULL;

588     for (i = 0; i < CONN_G_HASH_SIZE; i++) {

```

```

589         ASSERT(ipst->ips_ipcl_globalhash_fanout[i].connf_head == NULL);
590         mutex_destroy(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
591     }
592     kmem_free(ipst->ips_ipcl_globalhash_fanout,
593             sizeof (connf_t) * CONN_G_HASH_SIZE);
594     ipst->ips_ipcl_globalhash_fanout = NULL;

596     for (i = 0; i < ipst->ips_ipcl_dccp_fanout_size; i++) {
597         ASSERT(ipst->ips_ipcl_dccp_fanout[i].connf_head == NULL);
598         mutex_destroy(&ipst->ips_ipcl_dccp_fanout[i].connf_lock);
599     }
600     kmem_free(ipst->ips_ipcl_dccp_fanout, ipst->ips_ipcl_dccp_fanout_size *
601             sizeof (connf_t));
602     ipst->ips_ipcl_dccp_fanout = NULL;

604 #endif /* ! codereview */
605     ASSERT(ipst->ips_rts_clients->connf_head == NULL);
606     mutex_destroy(&ipst->ips_rts_clients->connf_lock);
607     kmem_free(ipst->ips_rts_clients, sizeof (connf_t));
608     ipst->ips_rts_clients = NULL;
609 }

611 /*
612  * conn creation routine. initialize the conn, sets the reference
613  * and inserts it in the global hash table.
614  */
615 conn_t *
616 ipcl_conn_create(uint32_t type, int sleep, netstack_t *ns)
617 {
618     conn_t *connp;
619     struct kmem_cache *conn_cache;

621     switch (type) {
622     case IPCL_SCTPCONN:
623         if ((connp = kmem_cache_alloc(sctp_conn_cache, sleep)) == NULL)
624             return (NULL);
625         sctp_conn_init(connp);
626         netstack_hold(ns);
627         connp->conn_netstack = ns;
628         connp->conn_ixa->ixa_ipst = ns->netstack_ip;
629         connp->conn_ixa->ixa_conn_id = (long)connp;
630         ipcl_globalhash_insert(connp);
631         return (connp);

633     case IPCL_TCPCONN:
634         conn_cache = tcp_conn_cache;
635         break;

637     case IPCL_UDPCONN:
638         conn_cache = udp_conn_cache;
639         break;

641     case IPCL_RAWIPCONN:
642         conn_cache = rawip_conn_cache;
643         break;

645     case IPCL_RTSCONN:
646         conn_cache = rts_conn_cache;
647         break;

649     case IPCL_IPCONN:
650         conn_cache = ip_conn_cache;
651         break;

653     case IPCL_DCCPCONN:
654         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_iocthref == 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         /* ix_a_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_timer_cache;
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_timer_cache = 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(&oconnp->conn_laddr_v6) ||
1084              IN6_IS_ADDR_UNSPECIFIED(&oconnp->conn_laddr_v6) ||
1085              IN6_IS_ADDR_V4MAPPED_ANY(&oconnp->conn_laddr_v6) ||
1086              IN6_IS_ADDR_V4MAPPED_ANY(&oconnp->conn_laddr_v6)) ||
1087              IN6_IS_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 adrs, 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 adrs, 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;

```



```

1315 #endif /* ! codereview */
1316     }
1317 }
1318
1319 #endif /* ! codereview */
1320     return (ret);
1321 }
1322
1323 int
1324 ipcl_bind_insert_v6(conn_t *connp)
1325 {
1326     connf_t      *connfp;
1327     int          ret = 0;
1328     ip_stack_t   *ipst = connp->conn_netstack->netstack_ip;
1329     uint16_t     lport = connp->conn_lport;
1330     uint8_t      protocol = connp->conn_proto;
1331
1332     if (IPCL_IS_IPTUN(connp)) {
1333         return (ipcl_iptun_hash_insert_v6(connp, ipst));
1334     }
1335
1336     switch (protocol) {
1337     default:
1338         if (is_system_labeled() &&
1339             check_exempt_conflict_v6(connp, ipst))
1340             return (EADDRINUSE);
1341         /* FALLTHROUGH */
1342     case IPPROTO_UDP:
1343         if (protocol == IPPROTO_UDP) {
1344             connfp = &ipst->ips_ipcl_udp_fanout[
1345                 IPCL_UDP_HASH(lport, ipst)];
1346         } else {
1347             connfp = &ipst->ips_ipcl_proto_fanout_v6[protocol];
1348         }
1349
1350         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6)) {
1351             IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1352         } else if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
1353             IPCL_HASH_INSERT_BOUND(connfp, connp);
1354         } else {
1355             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1356         }
1357         break;
1358
1359     case IPPROTO_TCP:
1360         /* Insert it in the Bind Hash */
1361         ASSERT(connp->conn_zoneid != ALL_ZONES);
1362         connfp = &ipst->ips_ipcl_bind_fanout[
1363             IPCL_BIND_HASH(lport, ipst)];
1364         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
1365             IPCL_HASH_INSERT_BOUND(connfp, connp);
1366         } else {
1367             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1368         }
1369         if (cl_inet_listen != NULL) {
1370             sa_family_t   addr_family;
1371             uint8_t       *laddrp;
1372
1373             if (connp->conn_ipversion == IPV6_VERSION) {
1374                 addr_family = AF_INET6;
1375                 laddrp =
1376                     (uint8_t *)&connp->conn_bound_addr_v6;
1377             } else {
1378                 addr_family = AF_INET;
1379                 laddrp = (uint8_t *)&connp->conn_bound_addr_v4;
1380             }

```

```

1381         connp->conn_flags |= IPCL_CL_LISTENER;
1382         (*cl_inet_listen)(
1383             connp->conn_netstack->netstack_stackid,
1384             IPPROTO_TCP, addr_family, laddrp, lport, NULL);
1385     }
1386     break;
1387
1388     case IPPROTO_SCTP:
1389         ret = ipcl_sctp_hash_insert(connp, lport);
1390         break;
1391
1392     case IPPROTO_DCCP:
1393         /* XXX:DCCP */
1394         break;
1395 #endif /* ! codereview */
1396     }
1397
1398     return (ret);
1399 }
1400
1401 /*
1402 * ipcl_conn_hash insertion routines.
1403 * The caller has already set conn_proto and the addresses/ports in the conn_t.
1404 */
1405
1406 int
1407 ipcl_conn_insert(conn_t *connp)
1408 {
1409     if (connp->conn_ipversion == IPV6_VERSION)
1410         return (ipcl_conn_insert_v6(connp));
1411     else
1412         return (ipcl_conn_insert_v4(connp));
1413 }
1414
1415 int
1416 ipcl_conn_insert_v4(conn_t *connp)
1417 {
1418     connf_t      *connfp;
1419     conn_t       *tconnp;
1420     int          ret = 0;
1421     ip_stack_t   *ipst = connp->conn_netstack->netstack_ip;
1422     uint16_t     lport = connp->conn_lport;
1423     uint8_t      protocol = connp->conn_proto;
1424
1425     if (IPCL_IS_IPTUN(connp))
1426         return (ipcl_iptun_hash_insert(connp, ipst));
1427
1428     switch (protocol) {
1429     case IPPROTO_TCP:
1430         /*
1431          * For TCP, we check whether the connection tuple already
1432          * exists before allowing the connection to proceed. We
1433          * also allow indexing on the zoneid. This is to allow
1434          * multiple shared stack zones to have the same tcp
1435          * connection tuple. In practice this only happens for
1436          * INADDR_LOOPBACK as it's the only local address which
1437          * doesn't have to be unique.
1438          */
1439         connfp = &ipst->ips_ipcl_conn_fanout[
1440             IPCL_CONN_HASH(connp->conn_faddr_v4,
1441                 connp->conn_ports, ipst)];
1442         mutex_enter(&connfp->connf_lock);
1443         for (tconnp = connfp->connf_head; tconnp != NULL;
1444             tconnp = tconnp->conn_next) {
1445             if (IPCL_CONN_MATCH(tconnp, connp->conn_proto,
1446                 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, "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);
1487     IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1488     mutex_exit(&connfp->connf_lock);
1489     /* XXX:DCCP */
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_rcv_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 tnrx.
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_connf->connf_lock);
1716             return (connp);
1717         }
1718
1719         mutex_exit(&bind_connf->connf_lock);
1720         break;
1721
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         }
1737
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         }
1746
1747         if (connp != NULL) {
1748             CONN_INC_REF(connp);
1749             mutex_exit(&connfp->connf_lock);
1750             return (connp);
1751         }
1752
1753         /*
1754          * We shouldn't come here for multicast/broadcast packets
1755          */
1756         mutex_exit(&connfp->connf_lock);
1757
1758         break;
1759
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         }
1775
1776         if (connp != NULL) {

```

```

1777             CONN_INC_REF(connp);
1778             mutex_exit(&connfp->connf_lock);
1779             return (connp);
1780         }
1781
1782         mutex_exit(&connfp->connf_lock);
1783         break;
1784
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     }
1791
1792     return (NULL);
1793 }
1794
1795 conn_t *
1796 ipcl_classify_v6(mblk_t *mp, uint8_t protocol, uint_t hdr_len,
1797     ip_rcv_attr_t *ira, ip_stack_t *ipst)
1798 {
1799     ip6_t          *ip6h;
1800     connf_t        *connfp, *bind_connf;
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;
1808
1809     ip6h = (ip6_t *)mp->b_rptr;
1810
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;
1816
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         }
1832
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 listener 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_timer_cache = tcp_timermp_alloc(kmflags);
2067     if (tcp->tcp_timer_cache == 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_UDPCCONN;
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_UDPCCONN);
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_RAWIPCCONN;
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         ixarefrele(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         ixarefrele(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     connp->conn_ixa->ixa_refcnt = 1;
2329     connp->conn_ixa->ixa_protocol = connp->conn_proto;
2330     connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);

2332     return (0);
2333 }

2335 /* ARGSUSED */
2336 static void
2337 dccp_conn_destructor(void *buf, void *cdrarg)
2338 {
2339     itc_t *itc = (itc_t *)buf;
2340     conn_t *connp = &itc->itc_conn;
2341     dccp_t *dccp = (dccp_t *)&itc[1];

2343     ASSERT(connp->conn_flags & IPCL_DCCPCONN);
2344     ASSERT(dccp->dccp_connp == connp);
2345     ASSERT(connp->conn_dccp == dccp);

2347     mutex_destroy(&connp->conn_lock);
2348     cv_destroy(&connp->conn_cv);
2349     rw_destroy(&connp->conn_ilg_lock);

2351     if (connp->conn_ixa != NULL) {
2352         ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2353         ASSERT(connp->conn_ixa->ixa_ire == NULL);
2354         ASSERT(connp->conn_ixa->ixa_nce == NULL);

2356         ixarefrele(connp->conn_ixa);
2357     }
2358 }

2360 #endif /* ! codereview */
2361 /*
2362  * Called as part of ipcl_conn_destroy to assert and clear any pointers
2363  * in the conn_t.
2364  *
2365  * Below we list all the pointers in the conn_t as a documentation aid.
2366  * The ones that we can not ASSERT to be NULL are #ifdef'd out.
2367  * If you add any pointers to the conn_t please add an ASSERT here
2368  * and #ifdef it out if it can't be actually asserted to be NULL.
2369  * In any case, we bzero most of the conn_t at the end of the function.
2370  */

```



```

2371 void
2372 ipcl_conn_cleanup(conn_t *connp)
2373 {
2374     ip_xmit_attr_t *ixa;

2376     ASSERT(connp->conn_latch == NULL);
2377     ASSERT(connp->conn_latch_in_policy == NULL);
2378     ASSERT(connp->conn_latch_in_action == NULL);
2379 #ifdef notdef
2380     ASSERT(connp->conn_rq == NULL);
2381     ASSERT(connp->conn_wq == NULL);
2382 #endif
2383     ASSERT(connp->conn_cred == NULL);
2384     ASSERT(connp->conn_g_fanout == NULL);
2385     ASSERT(connp->conn_g_next == NULL);
2386     ASSERT(connp->conn_g_prev == NULL);
2387     ASSERT(connp->conn_policy == NULL);
2388     ASSERT(connp->conn_fanout == NULL);
2389     ASSERT(connp->conn_next == NULL);
2390     ASSERT(connp->conn_prev == NULL);
2391     ASSERT(connp->conn_oper_pending_ill == NULL);
2392     ASSERT(connp->conn_ilg == NULL);
2393     ASSERT(connp->conn_drain_next == NULL);
2394     ASSERT(connp->conn_drain_prev == NULL);
2395 #ifdef notdef
2396     /* conn_idl is not cleared when removed from idl list */
2397     ASSERT(connp->conn_idl == NULL);
2398 #endif
2399     ASSERT(connp->conn_ipsec_opt_mp == NULL);
2400 #ifdef notdef
2401     /* conn_netstack is cleared by the caller; needed by ixa_cleanup */
2402     ASSERT(connp->conn_netstack == NULL);
2403 #endif

2405     ASSERT(connp->conn_helper_info == NULL);
2406     ASSERT(connp->conn_ixa != NULL);
2407     ixa = connp->conn_ixa;
2408     ASSERT(ixa->ixa_refcnt == 1);
2409     /* Need to preserve ixa_protocol */
2410     ixa_cleanup(ixa);
2411     ixa->ixa_flags = 0;

2413     /* Clear out the conn_t fields that are not preserved */
2414     bzero(&connp->conn_start_clr,
2415          sizeof(conn_t) -
2416          ((uchar_t *)&connp->conn_start_clr - (uchar_t *)connp));
2417 }

2419 /*
2420  * All conns are inserted in a global multi-list for the benefit of
2421  * walkers. The walk is guaranteed to walk all open conns at the time
2422  * of the start of the walk exactly once. This property is needed to
2423  * achieve some cleanups during unplumb of interfaces. This is achieved
2424  * as follows.
2425  *
2426  * ipcl_conn_create and ipcl_conn_destroy are the only functions that
2427  * call the insert and delete functions below at creation and deletion
2428  * time respectively. The conn never moves or changes its position in this
2429  * multi-list during its lifetime. CONN_CONDEMNED ensures that the refcnt
2430  * won't increase due to walkers, once the conn deletion has started. Note
2431  * that we can't remove the conn from the global list and then wait for
2432  * the refcnt to drop to zero, since walkers would then see a truncated
2433  * list. CONN_INCIPIENT ensures that walkers don't start looking at
2434  * conns until ip_open is ready to make them globally visible.
2435  * The global round robin multi-list locks are held only to get the
2436  * next member/insertion/deletion and contention should be negligible

```

```

2437  * if the multi-list is much greater than the number of cpus.
2438  */
2439 void
2440 ipcl_globalhash_insert(conn_t *connp)
2441 {
2442     int index;
2443     struct connf_s *connfp;
2444     ip_stack_t *ipst = connp->conn_netstack->netstack_ip;

2446     /*
2447      * No need for atomic here. Approximate even distribution
2448      * in the global lists is sufficient.
2449      */
2450     ipst->ips_conn_g_index++;
2451     index = ipst->ips_conn_g_index & (CONN_G_HASH_SIZE - 1);

2453     connp->conn_g_prev = NULL;
2454     /*
2455      * Mark as INCIPIENT, so that walkers will ignore this
2456      * for now, till ip_open is ready to make it visible globally.
2457      */
2458     connp->conn_state_flags |= CONN_INCIPIENT;

2460     connfp = &ipst->ips_ipcl_globalhash_fanout[index];
2461     /* Insert at the head of the list */
2462     mutex_enter(&connfp->connf_lock);
2463     connp->conn_g_next = connfp->connf_head;
2464     if (connp->conn_g_next != NULL)
2465         connp->conn_g_next->conn_g_prev = connp;
2466     connfp->connf_head = connp;

2468     /* The fanout bucket this conn points to */
2469     connp->conn_g_fanout = connfp;

2471     mutex_exit(&connfp->connf_lock);
2472 }

2474 void
2475 ipcl_globalhash_remove(conn_t *connp)
2476 {
2477     struct connf_s *connfp;

2479     /*
2480      * We were never inserted in the global multi list.
2481      * IPCL_NONE variety is never inserted in the global multilist
2482      * since it is presumed to not need any cleanup and is transient.
2483      */
2484     if (connp->conn_g_fanout == NULL)
2485         return;

2487     connfp = connp->conn_g_fanout;
2488     mutex_enter(&connfp->connf_lock);
2489     if (connp->conn_g_prev != NULL)
2490         connp->conn_g_prev->conn_g_next = connp->conn_g_next;
2491     else
2492         connfp->connf_head = connp->conn_g_next;
2493     if (connp->conn_g_next != NULL)
2494         connp->conn_g_next->conn_g_prev = connp->conn_g_prev;
2495     mutex_exit(&connfp->connf_lock);

2497     /* Better to stumble on a null pointer than to corrupt memory */
2498     connp->conn_g_next = NULL;
2499     connp->conn_g_prev = NULL;
2500     connp->conn_g_fanout = NULL;
2501 }

```

```

2503 /*
2504  * Walk the list of all conn_t's in the system, calling the function provided
2505  * With the specified argument for each.
2506  * Applies to both IPv4 and IPv6.
2507  *
2508  * CONNs may hold pointers to ill (conn_dhcpinit_ill and
2509  * conn_oper_pending_ill). To guard against stale pointers
2510  * ipcl_walk() is called to cleanup the conn_t's, typically when an interface is
2511  * unplumbed or removed. New conn_t's that are created while we are walking
2512  * may be missed by this walk, because they are not necessarily inserted
2513  * at the tail of the list. They are new conn_t's and thus don't have any
2514  * stale pointers. The CONN_CLOSING flag ensures that no new reference
2515  * is created to the struct that is going away.
2516  */
2517 void
2518 ipcl_walk(pfv_t func, void *arg, ip_stack_t *ipst)
2519 {
2520     int i;
2521     conn_t *connp;
2522     conn_t *prev_connp;
2523
2524     for (i = 0; i < CONN_G_HASH_SIZE; i++) {
2525         mutex_enter(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2526         prev_connp = NULL;
2527         connp = ipst->ips_ipcl_globalhash_fanout[i].connf_head;
2528         while (connp != NULL) {
2529             mutex_enter(&connp->conn_lock);
2530             if (connp->conn_state_flags &
2531                 (CONN_CONDEMNED | CONN_INCIPIENT)) {
2532                 mutex_exit(&connp->conn_lock);
2533                 connp = connp->conn_g_next;
2534                 continue;
2535             }
2536             CONN_INC_REF_LOCKED(connp);
2537             mutex_exit(&connp->conn_lock);
2538             mutex_exit(
2539                 &ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2540             (*func)(connp, arg);
2541             if (prev_connp != NULL)
2542                 CONN_DEC_REF(prev_connp);
2543             mutex_enter(
2544                 &ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2545             prev_connp = connp;
2546             connp = connp->conn_g_next;
2547         }
2548         mutex_exit(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2549         if (prev_connp != NULL)
2550             CONN_DEC_REF(prev_connp);
2551     }
2552 }
2553
2554 /*
2555  * Search for a peer TCP/IPv4 loopback conn by doing a reverse lookup on
2556  * the {src, dst, lport, fport} quadruplet. Returns with conn reference
2557  * held; caller must call CONN_DEC_REF. Only checks for connected entries
2558  * (peer tcp in ESTABLISHED state).
2559  */
2560 conn_t *
2561 ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *connp, ipha_t *ipha, tcpha_t *tcpha,
2562     ip_stack_t *ipst)
2563 {
2564     uint32_t ports;
2565     uint16_t *pports = (uint16_t *)&ports;
2566     connf_t *connfp;
2567     conn_t *tconnp;
2568     boolean_t zone_chk;

```

```

2570     /*
2571     * If either the source of destination address is loopback, then
2572     * both endpoints must be in the same Zone. Otherwise, both of
2573     * the addresses are system-wide unique (tcp is in ESTABLISHED
2574     * state) and the endpoints may reside in different Zones.
2575     */
2576     zone_chk = (ipha->ipha_src == htonl(INADDR_LOOPBACK) ||
2577         ipha->ipha_dst == htonl(INADDR_LOOPBACK));
2578
2579     pports[0] = tcpha->tha_fport;
2580     pports[1] = tcpha->tha_lport;
2581
2582     connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ipha->ipha_dst,
2583         ports, ipst)];
2584
2585     mutex_enter(&connfp->connf_lock);
2586     for (tconnp = connfp->connf_head; tconnp != NULL;
2587         tconnp = tconnp->conn_next) {
2588
2589         if (IPCL_CONN_MATCH(tconnp, IPPROTO_TCP,
2590             ipha->ipha_dst, ipha->ipha_src, ports) &&
2591             tconnp->conn_tcp->tcp_state == TCPS_ESTABLISHED &&
2592             (!zone_chk || tconnp->conn_zoneid == connp->conn_zoneid)) {
2593
2594             ASSERT(tconnp != connp);
2595             CONN_INC_REF(tconnp);
2596             mutex_exit(&connfp->connf_lock);
2597             return (tconnp);
2598         }
2599     }
2600     mutex_exit(&connfp->connf_lock);
2601     return (NULL);
2602 }
2603
2604 /*
2605  * Search for a peer TCP/IPv6 loopback conn by doing a reverse lookup on
2606  * the {src, dst, lport, fport} quadruplet. Returns with conn reference
2607  * held; caller must call CONN_DEC_REF. Only checks for connected entries
2608  * (peer tcp in ESTABLISHED state).
2609  */
2610 conn_t *
2611 ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *connp, ip6_t *ip6h, tcpha_t *tcpha,
2612     ip_stack_t *ipst)
2613 {
2614     uint32_t ports;
2615     uint16_t *pports = (uint16_t *)&ports;
2616     connf_t *connfp;
2617     conn_t *tconnp;
2618     boolean_t zone_chk;
2619
2620     /*
2621     * If either the source of destination address is loopback, then
2622     * both endpoints must be in the same Zone. Otherwise, both of
2623     * the addresses are system-wide unique (tcp is in ESTABLISHED
2624     * state) and the endpoints may reside in different Zones. We
2625     * don't do Zone check for link local address(es) because the
2626     * current Zone implementation treats each link local address as
2627     * being unique per system node, i.e. they belong to global Zone.
2628     */
2629     zone_chk = (IN6_IS_ADDR_LOOPBACK(&ip6h->ip6_src) ||
2630         IN6_IS_ADDR_LOOPBACK(&ip6h->ip6_dst));
2631
2632     pports[0] = tcpha->tha_fport;
2633     pports[1] = tcpha->tha_lport;

```

```

2635 connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_dst,
2636 ports, ipst)];

2638 mutex_enter(&connfp->connf_lock);
2639 for (tconnp = connfp->connf_head; tconnp != NULL;
2640 tconnp = tconnp->conn_next) {

2642     /* We skip conn_bound_if check here as this is loopback tcp */
2643     if (IPCL_CONN_MATCH_V6(tconnp, IPPROTO_TCP,
2644 ip6h->ip6_dst, ip6h->ip6_src, ports) &&
2645 tconnp->conn_tcp->tcp_state == TCPS_ESTABLISHED &&
2646 (!zone_chk || tconnp->conn_zoneid == connp->conn_zoneid)) {

2648         ASSERT(tconnp != connp);
2649         CONN_INC_REF(tconnp);
2650         mutex_exit(&connfp->connf_lock);
2651         return (tconnp);
2652     }
2653 }
2654 mutex_exit(&connfp->connf_lock);
2655 return (NULL);
2656 }

2658 /*
2659 * Find an exact {src, dst, lport, fport} match for a bounced datagram.
2660 * Returns with conn reference held. Caller must call CONN_DEC_REF.
2661 * Only checks for connected entries i.e. no INADDR_ANY checks.
2662 */
2663 conn_t *
2664 ipcl_tcp_lookup_reversed_ipv4(ip4_t *ip4a, tcpha_t *tcpha, int min_state,
2665 ip_stack_t *ipst)
2666 {
2667     uint32_t ports;
2668     uint16_t *pports;
2669     connf_t *connfp;
2670     conn_t *tconnp;

2672     pports = (uint16_t *)&ports;
2673     pports[0] = tcpha->tha_fport;
2674     pports[1] = tcpha->tha_lport;

2676     connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ip4a->ip4a_dst,
2677 ports, ipst)];

2679     mutex_enter(&connfp->connf_lock);
2680     for (tconnp = connfp->connf_head; tconnp != NULL;
2681 tconnp = tconnp->conn_next) {

2683         if (IPCL_CONN_MATCH(tconnp, IPPROTO_TCP,
2684 ip4a->ip4a_dst, ip4a->ip4a_src, ports) &&
2685 tconnp->conn_tcp->tcp_state >= min_state) {

2687             CONN_INC_REF(tconnp);
2688             mutex_exit(&connfp->connf_lock);
2689             return (tconnp);
2690         }
2691     }
2692     mutex_exit(&connfp->connf_lock);
2693     return (NULL);
2694 }

2696 /*
2697 * Find an exact {src, dst, lport, fport} match for a bounced datagram.
2698 * Returns with conn reference held. Caller must call CONN_DEC_REF.
2699 * Only checks for connected entries i.e. no INADDR_ANY checks.
2700 * Match on ifindex in addition to addresses.

```

```

2701 */
2702 conn_t *
2703 ipcl_tcp_lookup_reversed_ipv6(ip6_t *ip6h, tcpha_t *tcpha, int min_state,
2704 uint_t ifindex, ip_stack_t *ipst)
2705 {
2706     tcp_t *tcp;
2707     uint32_t ports;
2708     uint16_t *pports;
2709     connf_t *connfp;
2710     conn_t *tconnp;

2712     pports = (uint16_t *)&ports;
2713     pports[0] = tcpha->tha_fport;
2714     pports[1] = tcpha->tha_lport;

2716     connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_dst,
2717 ports, ipst)];

2719     mutex_enter(&connfp->connf_lock);
2720     for (tconnp = connfp->connf_head; tconnp != NULL;
2721 tconnp = tconnp->conn_next) {

2723         tcp = tconnp->conn_tcp;
2724         if (IPCL_CONN_MATCH_V6(tconnp, IPPROTO_TCP,
2725 ip6h->ip6_dst, ip6h->ip6_src, ports) &&
2726 tcp->tcp_state >= min_state &&
2727 (tconnp->conn_bound_if == 0 ||
2728 tconnp->conn_bound_if == ifindex)) {

2730             CONN_INC_REF(tconnp);
2731             mutex_exit(&connfp->connf_lock);
2732             return (tconnp);
2733         }
2734     }
2735     mutex_exit(&connfp->connf_lock);
2736     return (NULL);
2737 }

2739 /*
2740 * Finds a TCP/IPV4 listening connection; called by tcp_disconnect to locate
2741 * a listener when changing state.
2742 */
2743 conn_t *
2744 ipcl_lookup_listener_v4(uint16_t lport, ipaddr_t laddr, zoneid_t zoneid,
2745 ip_stack_t *ipst)
2746 {
2747     connf_t *bind_connfp;
2748     conn_t *connp;
2749     tcp_t *tcp;

2751     /*
2752     * Avoid false matches for packets sent to an IP destination of
2753     * all zeros.
2754     */
2755     if (laddr == 0)
2756         return (NULL);

2758     ASSERT(zoneid != ALL_ZONES);

2760     bind_connfp = &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
2761     mutex_enter(&bind_connfp->connf_lock);
2762     for (connp = bind_connfp->connf_head; connp != NULL;
2763 connp = connp->conn_next) {
2764         tcp = connp->conn_tcp;
2765         if (IPCL_BIND_MATCH(connp, IPPROTO_TCP, laddr, lport) &&
2766 IPCL_ZONE_MATCH(connp, zoneid) &&

```

```

2767         (tcp->tcp_listener == NULL)) {
2768             CONN_INC_REF(connp);
2769             mutex_exit(&bind_connfp->connf_lock);
2770             return (connp);
2771         }
2772     }
2773     mutex_exit(&bind_connfp->connf_lock);
2774     return (NULL);
2775 }

2777 /*
2778  * Finds a TCP/IPv6 listening connection; called by tcp_disconnect to locate
2779  * a listener when changing state.
2780  */
2781 conn_t *
2782 ipcl_lookup_listener_v6(uint16_t lport, in6_addr_t *laddr, uint_t ifindex,
2783     zoneid_t zoneid, ip_stack_t *ipst)
2784 {
2785     connf_t     *bind_connfp;
2786     conn_t      *connp = NULL;
2787     tcp_t       *tcp;

2789     /*
2790      * Avoid false matches for packets sent to an IP destination of
2791      * all zeros.
2792      */
2793     if (IN6_IS_ADDR_UNSPECIFIED(laddr))
2794         return (NULL);

2796     ASSERT(zoneid != ALL_ZONES);

2798     bind_connfp = &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
2799     mutex_enter(&bind_connfp->connf_lock);
2800     for (connp = bind_connfp->connf_head; connp != NULL;
2801         connp = connp->conn_next) {
2802         tcp = connp->conn_tcp;
2803         if (IPCL_BIND_MATCH_V6(connp, IPPROTO_TCP, *laddr, lport) &&
2804             IPCL_ZONE_MATCH(connp, zoneid) &&
2805             (connp->conn_bound_if == 0 ||
2806             connp->conn_bound_if == ifindex) &&
2807             tcp->tcp_listener == NULL) {
2808             CONN_INC_REF(connp);
2809             mutex_exit(&bind_connfp->connf_lock);
2810             return (connp);
2811         }
2812     }
2813     mutex_exit(&bind_connfp->connf_lock);
2814     return (NULL);
2815 }

2817 /*
2818  * ipcl_get_next_conn
2819  * get the next entry in the conn global list
2820  * and put a reference on the next_conn.
2821  * decrement the reference on the current conn.
2822  *
2823  * This is an iterator based walker function that also provides for
2824  * some selection by the caller. It walks through the conn_hash bucket
2825  * searching for the next valid connp in the list, and selects connections
2826  * that are neither closed nor condemned. It also REFHOLDS the conn
2827  * thus ensuring that the conn exists when the caller uses the conn.
2828  */
2829 conn_t *
2830 ipcl_get_next_conn(connf_t *connfp, conn_t *connp, uint32_t conn_flags)
2831 {
2832     conn_t *next_connp;

```

```

2834     if (connfp == NULL)
2835         return (NULL);

2837     mutex_enter(&connfp->connf_lock);

2839     next_connp = (connp == NULL) ?
2840         connfp->connf_head : connp->conn_g_next;

2842     while (next_connp != NULL) {
2843         mutex_enter(&next_connp->conn_lock);
2844         if (!(next_connp->conn_flags & conn_flags) ||
2845             (next_connp->conn_state_flags &
2846             (CONN_CONDEMNED | CONN_INCIPIENT))) {
2847             /*
2848              * This conn has been condemned or
2849              * is closing, or the flags don't match
2850              */
2851             mutex_exit(&next_connp->conn_lock);
2852             next_connp = next_connp->conn_g_next;
2853             continue;
2854         }
2855         CONN_INC_REF_LOCKED(next_connp);
2856         mutex_exit(&next_connp->conn_lock);
2857         break;
2858     }

2860     mutex_exit(&connfp->connf_lock);

2862     if (connp != NULL)
2863         CONN_DEC_REF(connp);

2865     return (next_connp);
2866 }

2868 #ifdef CONN_DEBUG
2869 /*
2870  * Trace of the last NBUF refhold/refrele
2871  */
2872 int
2873 conn_trace_ref(conn_t *connp)
2874 {
2875     int     last;
2876     conn_trace_t *ctb;

2878     ASSERT(MUTEX_HELD(&connp->conn_lock));
2879     last = connp->conn_trace_last;
2880     last++;
2881     if (last == CONN_TRACE_MAX)
2882         last = 0;

2884     ctb = &connp->conn_trace_buf[last];
2885     ctb->ctb_depth = getpcstack(ctb->ctb_stack, CONN_STACK_DEPTH);
2886     connp->conn_trace_last = last;
2887     return (1);
2888 }

2890 int
2891 conn_untrace_ref(conn_t *connp)
2892 {
2893     int     last;
2894     conn_trace_t *ctb;

2896     ASSERT(MUTEX_HELD(&connp->conn_lock));
2897     last = connp->conn_trace_last;
2898     last++;

```

```
2899     if (last == CONN_TRACE_MAX)
2900         last = 0;
2902     ctb = &connp->conn_trace_buf[last];
2903     ctb->ctb_depth = getpcstack(ctb->ctb_stack, CONN_STACK_DEPTH);
2904     connp->conn_trace_last = last;
2905     return (1);
2906 }
2907 #endif
```

new/usr/src/uts/common/inet/ip_impl.h

1

```
*****
6502 Wed Jun 13 12:05:04 2012
new/usr/src/uts/common/inet/ip_impl.h
%B
*****
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 */
21 /*
22 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #ifndef _INET_IP_IMPL_H
27 #define _INET_IP_IMPL_H

29 /*
30 * IP implementation private declarations. These interfaces are
31 * used to build the IP module and are not meant to be accessed
32 * by any modules except IP itself. They are undocumented and are
33 * subject to change without notice.
34 */

36 #ifdef __cplusplus
37 extern "C" {
38 #endif

40 #ifdef _KERNEL

42 #include <sys/sdt.h>
43 #include <sys/dld.h>
44 #include <inet/tunables.h>

46 #define IP_MOD_ID          5701

48 #define INET_NAME          "ip"

50 #ifdef _BIG_ENDIAN
51 #define IP_HDR_CSUM_TTL_ADJUST 256
52 #define IP_TCP_CSUM_COMP    IPPROTO_TCP
53 #define IP_UDP_CSUM_COMP   IPPROTO_UDP
54 #define IP_ICMPV6_CSUM_COMP IPPROTO_ICMPV6
55 #define IP_DCCP_CSUM_COMP  IPPROTO_DCCP
56 #endif /* ! codereview */
57 #else
58 #define IP_HDR_CSUM_TTL_ADJUST 1
59 #define IP_TCP_CSUM_COMP      (IPPROTO_TCP << 8)
60 #define IP_UDP_CSUM_COMP      (IPPROTO_UDP << 8)
61 #define IP_ICMPV6_CSUM_COMP  (IPPROTO_ICMPV6 << 8)
```

new/usr/src/uts/common/inet/ip_impl.h

2

```
62 #define IP_DCCP_CSUM_COMP      (IPPROTO_DCCP << 8)
63 #endif /* ! codereview */
64 #endif

66 #define TCP_CHECKSUM_OFFSET    16
67 #define TCP_CHECKSUM_SIZE      2

69 #define UDP_CHECKSUM_OFFSET    6
70 #define UDP_CHECKSUM_SIZE      2

72 #define ICMPV6_CHECKSUM_OFFSET 2
73 #define ICMPV6_CHECKSUM_SIZE  2

75 #define DCCP_CHECKSUM_OFFSET    6
76 #define DCCP_CHECKSUM_SIZE      2

78 #endif /* ! codereview */
79 #define IPH_TCPH_CHECKSUMP(ipha, hlen) \
80     ((uint16_t *)(((uchar_t *) (ipha)) + ((hlen) + TCP_CHECKSUM_OFFSET)))

82 #define IPH_UDPH_CHECKSUMP(ipha, hlen) \
83     ((uint16_t *)(((uchar_t *) (ipha)) + ((hlen) + UDP_CHECKSUM_OFFSET)))

85 #define IPH_ICMPV6_CHECKSUMP(ipha, hlen) \
86     ((uint16_t *)(((uchar_t *) (ipha)) + ((hlen) + ICMPV6_CHECKSUM_OFFSET)))

88 #define IPH_DCCPH_CHECKSUMP(ipha, hlen) \
89     ((uint16_t *)(((uchar_t *) (ipha)) + ((hlen) + DCCP_CHECKSUM_OFFSET)))

91 #endif /* ! codereview */
92 #define ILL_HCKSUM_CAPABLE(ill) \
93     (((ill)->ill_capabilities & ILL_CAPAB_HCKSUM) != 0)

95 /*
96 * Macro to adjust a given checksum value depending on any prepended
97 * or postpended data on the packet. It expects the start offset to
98 * begin at an even boundary and that the packet consists of at most
99 * two mblks.
100 */
101 #define IP_ADJCKSUM_PARTIAL(cksum_start, mp, mpl, len, adj) { \
102     /* \
103     * Prepended extraneous data; adjust checksum. \
104     */ \
105     if ((len) > 0) \
106         (adj) = IP_BCSUM_PARTIAL(cksum_start, len, 0); \
107     else \
108         (adj) = 0; \
109     /* \
110     * len is now the total length of mblk(s) \
111     */ \
112     (len) = MBLKL(mp); \
113     if ((mpl) == NULL) \
114         (mpl) = (mp); \
115     else \
116         (len) += MBLKL(mpl); \
117     /* \
118     * Postpended extraneous data; adjust checksum. \
119     */ \
120     if (((len) = (DB_CKSUMEND(mp) - len)) > 0) { \
121         uint32_t _pad; \
122         \
123         _pad = IP_BCSUM_PARTIAL((mpl)->b_wptr, len, 0); \
124         /* \
125         * If the postpended extraneous data was odd \
126         * byte aligned, swap resulting checksum bytes. \
127         */ \
```

```

128     if ((uintptr_t)(mpl)->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 Wed Jun 13 12:05:06 2012
new/usr/src/uts/common/inet/ip_stack.h
%B
*****
_____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 IPV6_MASK_TABLE_SIZE (IPV6_ABITS + 1) /* 129 ptrs */
238 struct irb *ips_ip_forwarding_table_v6[IPV6_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_mrouter 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 mfc_b   *ips_mfcs; /* kernel routing table */
338 struct tbf     *ips_tbf;
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 */

```

```

*****
26769 Wed Jun 13 12:05:10 2012
new/usr/src/uts/common/inet/ipclassifier.h
%B
*****
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 */
21 /*
22 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #ifndef _INET_IPCLASSIFIER_H
27 #define _INET_IPCLASSIFIER_H

29 #ifdef __cplusplus
30 extern "C" {
31 #endif

33 #include <inet/common.h>
34 #include <inet/ip.h>
35 #include <inet/mi.h>
36 #include <inet/tcp.h>
37 #include <inet/ip6.h>
38 #include <netinet/in.h> /* for IPPROTO_* constants */
39 #include <sys/sdt.h>
40 #include <sys/socket_proto.h>
41 #include <sys/sunddi.h>
42 #include <sys/sunldi.h>

44 typedef void (*edesc_rpf)(void *, mblk_t *, void *, ip_recv_attr_t *);
45 struct icmph_s;
46 struct icmp6_hdr;
47 typedef boolean_t (*edesc_vpf)(conn_t *, void *, struct icmph_s *,
48     struct icmp6_hdr *, ip_recv_attr_t *);

50 /*
51 * =====
52 * = The CONNECTION =
53 * =====
54 */

56 /*
57 * The connection structure contains the common information/flags/ref needed.
58 * Implementation will keep the connection struct, the layers (with their
59 * respective data for event i.e. tcp_t if event was tcp_input_data) all in one
60 * contiguous memory location.
61 */

```

```

63 /* Conn Flags */
64 /* Unused 0x00020000 */
65 /* Unused 0x00040000 */
66 #define IPCL_FULLY_BOUND 0x00080000 /* Bound to correct squeue */
67 /* Unused 0x00100000 */
68 /* Unused 0x00200000 */
69 /* Unused 0x00400000 */
70 #define IPCL_CL_LISTENER 0x00800000 /* Cluster listener */
71 /* Unused 0x01000000 */
72 /* Unused 0x02000000 */
73 /* Unused 0x04000000 */
74 /* Unused 0x08000000 */
75 /* Unused 0x10000000 */
76 /* Unused 0x20000000 */
77 #define IPCL_CONNECTED 0x40000000 /* Conn in connected table */
78 #define IPCL_BOUND 0x80000000 /* Conn in bind table */

80 /* Flags identifying the type of conn */
81 #define IPCL_TCPCONN 0x00000001 /* From tcp_conn_cache */
82 #define IPCL_SCTPCONN 0x00000002 /* From sctp_conn_cache */
83 #define IPCL_IPCCONN 0x00000004 /* From ip_conn_cache */
84 #define IPCL_UDPCONN 0x00000008 /* From udp_conn_cache */
85 #define IPCL_RAWIPCONN 0x00000010 /* From rawip_conn_cache */
86 #define IPCL_RTSCONN 0x00000020 /* From rts_conn_cache */
87 /* Unused 0x00000040 */
88 #define IPCL_IPTUN 0x00000080 /* iptun module above us */
89 #define IPCL_DCCPCONN 0x00000100 /* From dccp_conn_cache */
90 #endif /* ! codereview */

92 #define IPCL_NONSTR 0x00001000 /* A non-STREAMS socket */
93 /* Unused 0x10000000 */

95 #define IPCL_REMOVED 0x00000100
96 #define IPCL_REUSED 0x00000200

98 #define IPCL_IS_CONNECTED(connp) \
99     ((connp)->conn_flags & IPCL_CONNECTED)

101 #define IPCL_IS_BOUND(connp) \
102     ((connp)->conn_flags & IPCL_BOUND)

104 /*
105 * Can't use conn_proto since we need to tell difference
106 * between a real TCP socket and a SOCK_RAW, IPPROTO_TCP.
107 */
108 #define IPCL_IS_TCP(connp) \
109     ((connp)->conn_flags & IPCL_TCPCONN)

111 #define IPCL_IS_SCTP(connp) \
112     ((connp)->conn_flags & IPCL_SCTPCONN)

114 #define IPCL_IS_UDP(connp) \
115     ((connp)->conn_flags & IPCL_UDPCONN)

117 #define IPCL_IS_RAWIP(connp) \
118     ((connp)->conn_flags & IPCL_RAWIPCONN)

120 #define IPCL_IS_RTS(connp) \
121     ((connp)->conn_flags & IPCL_RTSCONN)

123 #define IPCL_IS_IPTUN(connp) \
124     ((connp)->conn_flags & IPCL_IPTUN)

126 #define IPCL_IS_DCCP(connp) \
127     ((connp)->conn_flags & IPCL_DCCPCONN)

```

```

129 #endif /* ! codereview */
130 #define IPCL_IS_NONSTR(connp) ((connp)->conn_flags & IPCL_NONSTR)

132 typedef struct connf_s connf_t;

134 typedef struct
135 {
136     int     ctb_depth;
137 #define CONN_STACK_DEPTH 15
138     pc_t    ctb_stack[CONN_STACK_DEPTH];
139 } conn_trace_t;

141 typedef struct ip_helper_minor_info_s {
142     dev_t    ip_minfo_dev;      /* Device */
143     vmem_t   *ip_minfo_arena;   /* Arena */
144 } ip_helper_minfo_t;

146 /*
147  * ip helper stream info
148  */
149 typedef struct ip_helper_stream_info_s {
150     ldi_handle_t    iphs_handle;
151     queue_t         *iphs_rq;
152     queue_t         *iphs_wq;
153     ip_helper_minfo_t    *iphs_minfo;
154 } ip_helper_stream_info_t;

156 /*
157  * Mandatory Access Control mode, in conn_t's conn_mac_mode field.
158  * CONN_MAC_DEFAULT: strict enforcement of MAC.
159  * CONN_MAC_AWARE: allows communications between unlabeled systems
160  *                 and privileged daemons
161  * CONN_MAC_IMPLICIT: allows communications without explicit labels
162  *                   on the wire with privileged daemons.
163  *
164  * CONN_MAC_IMPLICIT is intended specifically for labeled IPsec key management
165  * in networks which don't pass CIPSO-labeled packets.
166  */
167 #define CONN_MAC_DEFAULT 0
168 #define CONN_MAC_AWARE 1
169 #define CONN_MAC_IMPLICIT 2

171 /*
172  * conn receive ancillary definition.
173  *
174  * These are the set of socket options that make the receive side
175  * potentially pass up ancillary data items.
176  * We have a union with an integer so that we can quickly check whether
177  * any ancillary data items need to be added.
178  */
179 typedef struct crb_s {
180     union {
181         uint32_t    crbu_all;
182         struct {
183             uint32_t
184             crbb_recvdstaddr : 1,      /* IP_RECVDSTADDR option */
185             crbb_recvopts : 1,        /* IP_RECVOPTS option */
186             crbb_recvif : 1,          /* IP_RECVIF option */
187             crbb_recvslla : 1,        /* IP_RECVSLLA option */

188             crbb_recvttl : 1,         /* IP_RECVTTL option */
189             crbb_ip_recvpktinfo : 1,  /* IP*_RECVPKTINFO option */
190             crbb_ipv6_recvhoplimit : 1, /* IPV6_RECVHOPLIMIT option */
191             crbb_ipv6_recvhopopts : 1, /* IPV6_RECVHOPOPTS option */

```

```

194     crbb_ipv6_recvdstopts : 1,      /* IPV6_RECVDSTOPTS option */
195     crbb_ipv6_recvrthdr : 1,        /* IPV6_RECVRTHDR option */
196     crbb_old_ipv6_recvdstopts : 1,  /* old form of IPV6_DSTOPTS */
197     crbb_ipv6_recvrthdrdstopts : 1, /* IPV6_RECVRTHDRDSTOPTS */

199     crbb_ipv6_recvtclass : 1,      /* IPV6_RECVTCLASS */
200     crbb_recvucred : 1,            /* IP_RECVUCRED option */
201     crbb_timestamp : 1;           /* SO_TIMESTAMP "socket" option */

203     } crbb;
204 } crbu;
205 } crb_t;

207 #define crb_all                crbu.crbu_all
208 #define crb_recvdstaddr        crbu.crbbb.crbbb_recvdstaddr
209 #define crb_recvopts           crbu.crbbb.crbbb_recvopts
210 #define crb_recvif             crbu.crbbb.crbbb_recvif
211 #define crb_recvslla          crbu.crbbb.crbbb_recvslla
212 #define crb_recvttl           crbu.crbbb.crbbb_recvttl
213 #define crb_ip_recvpktinfo     crbu.crbbb.crbbb_ip_recvpktinfo
214 #define crb_ipv6_recvhoplimit  crbu.crbbb.crbbb_ipv6_recvhoplimit
215 #define crb_ipv6_recvhopopts  crbu.crbbb.crbbb_ipv6_recvhopopts
216 #define crb_ipv6_recvdstopts  crbu.crbbb.crbbb_ipv6_recvdstopts
217 #define crb_ipv6_recvrthdr     crbu.crbbb.crbbb_ipv6_recvrthdr
218 #define crb_old_ipv6_recvdstopts crbu.crbbb.crbbb_old_ipv6_recvdstopts
219 #define crb_ipv6_recvrthdrdstopts crbu.crbbb.crbbb_ipv6_recvrthdrdstopts
220 #define crb_ipv6_recvtclass    crbu.crbbb.crbbb_ipv6_recvtclass
221 #define crb_recvucred          crbu.crbbb.crbbb_recvucred
222 #define crb_timestamp          crbu.crbbb.crbbb_timestamp

224 /*
225  * The initial fields in the conn_t are setup by the kmem_cache constructor,
226  * and are preserved when it is freed. Fields after that are bzero'ed when
227  * the conn_t is freed.
228  *
229  * Much of the conn_t is protected by conn_lock.
230  *
231  * conn_lock is also used by some ULPs (like UDP and RAWIP) to protect
232  * their state.
233  */
234 struct conn_s {
235     kmutex_t    conn_lock;
236     uint32_t    conn_ref;          /* Reference counter */
237     uint32_t    conn_flags;      /* Conn Flags */

238     union {
239         tcp_t        *cp_tcp;      /* Pointer to the tcp struct */
240         struct udp_s *cp_udp;      /* Pointer to the udp struct */
241         struct icmp_s *cp_icmp;    /* Pointer to rawip struct */
242         struct rts_s *cp_rts;      /* Pointer to rts struct */
243         struct iptun_s *cp_iptun;  /* Pointer to iptun_t */
244         struct sctp_s *cp_sctp;    /* For IPCL SCTPCONN */
245         struct dccp_s *cp_dccp;    /* Pointer to dccp struct */

246     };
247 #endif /* ! codereview */
248     void          *cp_priv;
249     } conn_proto_priv;
250 #define conn_tcp        conn_proto_priv.cp_tcp
251 #define conn_udp        conn_proto_priv.cp_udp
252 #define conn_icmp      conn_proto_priv.cp_icmp
253 #define conn_rts       conn_proto_priv.cp_rts
254 #define conn_iptun     conn_proto_priv.cp_iptun
255 #define conn_sctp      conn_proto_priv.cp_sctp
256 #define conn_dccp      conn_proto_priv.cp_dccp
257 #endif /* ! codereview */
258 #define conn_priv      conn_proto_priv.cp_priv

```

```

260 kcondvar_t      conn_cv;
261 uint8_t         conn_proto;          /* protocol type */

263 edesc_rpf       conn_recv;          /* Pointer to recv routine */
264 edesc_rpf       conn_recvicmp;     /* For ICMP error */
265 edesc_vpf       conn_verifyicmp;   /* Verify ICMP error */

267 ip_xmit_attr_t  *conn_ixa;          /* Options if no ancil data */

269 /* Fields after this are bzero'ed when the conn_t is freed. */
270 #define conn_start_clr conn_recv_ancillary

272 /* Options for receive-side ancillary data */
273 crb_t           conn_recv_ancillary;

275 squeue_t        *conn_sqp;          /* Squeue for processing */
276 uint_t          conn_state_flags;   /* IP state flags */

278 int             conn_lingertime;    /* linger time (in seconds) */

280 unsigned int
281 conn_on_sqp : 1,          /* Conn is being processed */
282 conn_linger : 1,         /* SO_LINGER state */
283 conn_uselookback : 1,   /* SO_USELOOPBACK state */
284 conn_broadcast : 1,     /* SO_BROADCAST state */

286 conn_reuseaddr : 1,     /* SO_REUSEADDR state */
287 conn_keeplive : 1,     /* SO_KEEPAIVE state */
288 conn_multi_router : 1, /* Wants all multicast pkts */
289 conn_unspec_src : 1,   /* IP_UNSPEC_SRC */

291 conn_policy_cached : 1, /* Is policy cached/latched ? */
292 conn_in_enforce_policy : 1, /* Enforce Policy on inbound */
293 conn_out_enforce_policy : 1, /* Enforce Policy on outbound */
294 conn_debug : 1,        /* SO_DEBUG */

296 conn_ipv6_v6only : 1,  /* IPV6_V6ONLY */
297 conn_oobinline : 1,    /* SO_OOINLINE state */
298 conn_dgram_errind : 1, /* SO_DGRAM_ERRIND state */
299 conn_exclbind : 1,     /* SO_EXCLBIND state */

301 conn_mdt_ok : 1,       /* MDT is permitted */
302 conn_allzones : 1,     /* SO_ALLZONES */
303 conn_ipv6_recvpathmtu : 1, /* IPV6_RECVPATHMTU */
304 conn_mcbc_bind : 1,   /* Bound to multi/broadcast */

306 conn_pad_to_bit_31 : 12;

308 boolean_t       conn_blocked;       /* conn is flow-controlled */

310 squeue_t        *conn_initial_sqp;  /* Squeue at open time */
311 squeue_t        *conn_final_sqp;    /* Squeue after connect */
312 ill_t           *conn_dhcpinit_ill; /* IP_DHCPINIT_IF */
313 ipsec_latch_t   *conn_latch;        /* latched IDS */
314 struct ipsec_policy_s *conn_latch_in_policy; /* latched policy (in) */
315 struct ipsec_action_s *conn_latch_in_action; /* latched action (in) */
316 uint_t          conn_bound_if;      /* IP_BOUND_IF */
317 queue_t         *conn_rq;           /* Read queue */
318 queue_t         *conn_wq;           /* Write queue */
319 dev_t           conn_dev;           /* Minor number */
320 vmem_t          *conn_minor_arena;  /* Minor arena */
321 ip_helper_stream_info_t *conn_helper_info;

323 cred_t          *conn_cred;         /* Credentials */
324 pid_t           conn_cpuid;         /* pid from open/connect */
325 uint64_t        conn_open_time;     /* time when this was opened */

```

```

327 connf_t         *conn_g_fanout;     /* Global Hash bucket head */
328 struct conn_s   *conn_g_next;       /* Global Hash chain next */
329 struct conn_s   *conn_g_prev;       /* Global Hash chain prev */
330 struct ipsec_policy_head_s *conn_policy; /* Configured policy */
331 in6_addr_t      conn_bound_addr_v6; /* Address in bind() */
332 #define conn_bound_addr_v4 V4_PART_OF_V6(conn_bound_addr_v6)
333 connf_t         *conn_fanout;       /* Hash bucket we're part of */
334 struct conn_s   *conn_next;         /* Hash chain next */
335 struct conn_s   *conn_prev;         /* Hash chain prev */

337 struct {
338     in6_addr_t connua_laddr;         /* Local address - match */
339     in6_addr_t connua_faddr;        /* Remote address */
340 } connua_v6addr;
341 #define conn_laddr_v4 V4_PART_OF_V6(connua_v6addr.connua_laddr)
342 #define conn_faddr_v4 V4_PART_OF_V6(connua_v6addr.connua_faddr)
343 #define conn_laddr_v6 connua_v6addr.connua_laddr
344 #define conn_faddr_v6 connua_v6addr.connua_faddr
345 in6_addr_t      conn_saddr_v6;      /* Local address - source */
346 #define conn_saddr_v4 V4_PART_OF_V6(conn_saddr_v6)

348 union {
349     /* Used for classifier match performance */
350     uint32_t     connu_ports2;
351     struct {
352         in_port_t connu_fport;      /* Remote port */
353         in_port_t connu_lport;      /* Local port */
354     } connu_ports;
355 } u_port;
356 #define conn_fport u_port.connu_ports.connu_fport
357 #define conn_lport u_port.connu_ports.connu_lport
358 #define conn_ports u_port.connu_ports2

360 uint_t          conn_incoming_ifindex; /* IP{,V6}_BOUND_IF, scopeid */
361 ill_t           *conn_oper_pending_ill; /* pending shared ioctl */

363 krwlock_t       conn_ilg_lock;       /* Protects conn_ilg */
364 ill_t           *conn_ilg;           /* Group memberships */

366 kcondvar_t      conn_refcv;         /* For conn_oper_pending_ill */

368 struct conn_s   *conn_drain_next;    /* Next conn in drain list */
369 struct conn_s   *conn_drain_prev;    /* Prev conn in drain list */
370 idl_t           *conn_idl;           /* Ptr to the drain list head */
371 mblk_t          *conn_ipsec_opt_mp;  /* ipsec option mblk */
372 zoneid_t        conn_zoneid;         /* zone connection is in */
373 int             conn_rtaware;        /* RT_AWARE sockopt value */
374 kcondvar_t      conn_sq_cv;         /* For non-STREAMS socket IO */
375 sock_upcalls_t  *conn_upcalls;       /* Upcalls to socks */
376 sock_upper_handle_t conn_upper_handle; /* Upper handle: sonode */

378 unsigned int
379 conn_mlp_type : 2,          /* mlp_type_t; tsol/tndb.h */
380 conn_anon_mlp : 1,         /* user wants anon MLP */
381 conn_anon_port : 1,        /* user bound anonymously */

383 conn_mac_mode : 2,         /* normal/loose/implicit MAC */
384 conn_anon_priv_bind : 1,   /* * ANON_PRIV_BIND state */
385 conn_zone_is_global : 1,   /* GLOBAL_ZONEID */
386 conn_isvrrp : 1,          /* VRRP control socket */
387 conn_spare : 23;

389 boolean_t       conn_flow_cntrld;
390 netstack_t      *conn_netstack;     /* Corresponds to a netstack_hold */

```

```

392 /*
393  * IP format that packets received for this struct should use.
394  * Value can be IP4_VERSION or IPV6_VERSION.
395  * The sending version is encoded using IXAF_IS_IPV4.
396  */
397 ushort_t      conn_ipversion;

399 /* Written to only once at the time of opening the endpoint */
400 sa_family_t   conn_family;      /* Family from socket() call */
401 uint_t        conn_so_type;     /* Type from socket() call */

403 uint_t        conn_sndbuf;      /* SO_SNDBUF state */
404 uint_t        conn_rcvbuf;      /* SO_RCVBUF state */
405 uint_t        conn_wroff;       /* Current write offset */

407 uint_t        conn_sndlowat;    /* Send buffer low water mark */
408 uint_t        conn_rcvlowat;    /* Recv buffer low water mark */

410 uint8_t       conn_default_ttl; /* Default TTL/hoplimit */

412 uint32_t      conn_flowinfo; /* Connected flow id and tclass */

414 /*
415  * The most recent address for sendto. Initially set to zero
416  * which is always different than then the destination address
417  * since the send interprets zero as the loopback address.
418  */
419 in6_addr_t    conn_v6lastdst;
420 #define conn_v4lastdst V4_PART_OF_V6(conn_v6lastdst)
421 ushort_t     conn_lastipversion;
422 in_port_t    conn_lastdstport;
423 uint32_t     conn_lastflowinfo; /* IPv6-only */
424 uint_t       conn_lastscopeid; /* IPv6-only */
425 uint_t       conn_lastsrcid; /* Only for AF_INET6 */
426 /*
427  * When we are not connected conn_saddr might be unspecified.
428  * We track the source that was used with conn_v6lastdst here.
429  */
430 in6_addr_t    conn_v6lastsrc;
431 #define conn_v4lastsrc V4_PART_OF_V6(conn_v6lastsrc)

433 /* Templates for transmitting packets */
434 ip_pkt_t      conn_xmit_ipp; /* Options if no ancil data */

436 /*
437  * Header template - conn_ht_ulp is a pointer into conn_ht_iphc.
438  * Note that ixa_ip_hdr_length indicates the offset of ht_ulp in
439  * ht_iphc
440  *
441  * The header template is maintained for connected endpoints (and
442  * updated when sticky options are changed) and also for the lastdst.
443  * There is no conflict between those usages since SOCK_DGRAM and
444  * SOCK_RAW can not be used to specify a destination address (with
445  * sendto/sendmsg) if the socket has been connected.
446  */
447 uint8_t       *conn_ht_iphc; /* Start of IP header */
448 uint_t        conn_ht_iphc_allocated; /* Allocated buffer size */
449 uint_t        conn_ht_iphc_len; /* IP+ULP size */
450 uint8_t       *conn_ht_ulp; /* Upper-layer header */
451 uint_t        conn_ht_ulp_len; /* ULP header len */

453 /* Checksum to compensate for source routed packets. Host byte order */
454 uint32_t      conn_sum;

456 uint32_t      conn_ioctlref; /* ioctl ref count */
457 #ifndef CONN_DEBUG

```

```

458 #define CONN_TRACE_MAX 10
459 int           conn_trace_last; /* ndx of last used tracebuf */
460 conn_trace_t  conn_trace_buf[CONN_TRACE_MAX];
461 #endif
462 };

464 /*
465  * connf_t - connection fanout data.
466  *
467  * The hash tables and their linkage (conn_t.{hashnextp, hashprevp}) are
468  * protected by the per-bucket lock. Each conn_t inserted in the list
469  * points back at the connf_t that heads the bucket.
470  */
471 struct connf_s {
472     struct conn_s *connf_head;
473     kmutex_t      connf_lock;
474 };

476 #define CONN_INC_REF(connp) { \
477     mutex_enter(&(connp)->conn_lock); \
478     DTRACE_PROBE1(conn_inc_ref, conn_t *, connp); \
479     ASSERT(conn_trace_ref(connp)); \
480     (connp)->conn_ref++; \
481     ASSERT((connp)->conn_ref != 0); \
482     mutex_exit(&(connp)->conn_lock); \
483 }

485 #define CONN_INC_REF_LOCKED(connp) { \
486     DTRACE_PROBE1(conn_inc_ref, conn_t *, connp); \
487     ASSERT(MUTEX_HELD(&(connp)->conn_lock)); \
488     ASSERT(conn_trace_ref(connp)); \
489     (connp)->conn_ref++; \
490     ASSERT((connp)->conn_ref != 0); \
491 }

493 #define CONN_DEC_REF(connp) { \
494     mutex_enter(&(connp)->conn_lock); \
495     DTRACE_PROBE1(conn_dec_ref, conn_t *, connp); \
496     /* \
497      * The squeue framework always does a CONN_DEC_REF after return \
498      * from TCP. Hence the refcnt must be at least 2 if conn_on_sq \
499      * is B_TRUE and conn_ref is being decremented. This is to \
500      * account for the mblk being currently processed. \
501      */ \
502     if ((connp)->conn_ref == 0 || \
503         ((connp)->conn_ref == 1 && (connp)->conn_on_sq)) \
504         cmn_err(CE_PANIC, "CONN_DEC_REF: connp(%p) has ref " \
505             "= %d\n", (void *) (connp), (connp)->conn_ref); \
506     ASSERT(conn_untrace_ref(connp)); \
507     (connp)->conn_ref--; \
508     if ((connp)->conn_ref == 0) { \
509         /* Refcnt can't increase again, safe to drop lock */ \
510         mutex_exit(&(connp)->conn_lock); \
511         ipcl_conn_destroy(connp); \
512     } else { \
513         cv_broadcast(&(connp)->conn_cv); \
514         mutex_exit(&(connp)->conn_lock); \
515     } \
516 }

518 /*
519  * For use with subsystems within ip which use ALL_ZONES as a wildcard
520  */
521 #define IPCL_ZONEID(connp) \
522     ((connp)->conn_allzones ? ALL_ZONES : (connp)->conn_zoneid)

```

```

524 /*
525  * For matching between a conn_t and a zoneid.
526  */
527 #define IPCL_ZONE_MATCH(connp, zoneid) \
528     (((connp)->conn_allzones) || \
529      ((zoneid) == ALL_ZONES) || \
530      (connp)->conn_zoneid == (zoneid))
531
532 /*
533  * On a labeled system, we must treat bindings to ports
534  * on shared IP addresses by sockets with MAC exemption
535  * privilege as being in all zones, as there's
536  * otherwise no way to identify the right receiver.
537  */
538
539 #define IPCL_CONNS_MAC(conn1, conn2) \
540     (((conn1)->conn_mac_mode != CONN_MAC_DEFAULT) || \
541      ((conn2)->conn_mac_mode != CONN_MAC_DEFAULT))
542
543 #define IPCL_BIND_ZONE_MATCH(conn1, conn2) \
544     (IPCL_CONNS_MAC(conn1, conn2) || \
545      IPCL_ZONE_MATCH(conn1, conn2->conn_zoneid) || \
546      IPCL_ZONE_MATCH(conn2, conn1->conn_zoneid))
547
548
549 #define _IPCL_V4_MATCH(v6addr, v4addr) \
550     (V4_PART_OF_V6((v6addr)) == (v4addr) && IN6_IS_ADDR_V4MAPPED(&(v6addr)))
551
552 #define _IPCL_V4_MATCH_ANY(addr) \
553     (IN6_IS_ADDR_V4MAPPED_ANY(&(addr)) || IN6_IS_ADDR_UNSPECIFIED(&(addr)))
554
555
556 /*
557  * IPCL_PROTO_MATCH() and IPCL_PROTO_MATCH_V6() only matches conns with
558  * the specified ira_zoneid or conn_allzones by calling conn_wantpacket.
559  */
560 #define IPCL_PROTO_MATCH(connp, ira, ipha) \
561     (((connp)->conn_laddr_v4 == INADDR_ANY) || \
562      (((connp)->conn_laddr_v4 == ((ipha)->ipha_dst)) && \
563       ((connp)->conn_faddr_v4 == INADDR_ANY) || \
564       ((connp)->conn_faddr_v4 == ((ipha)->ipha_src)))) && \
565     conn_wantpacket((connp), (ira), (ipha))
566
567 #define IPCL_PROTO_MATCH_V6(connp, ira, ip6h) \
568     ((IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
569      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &((ip6h)->ip6_dst)) && \
570      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) || \
571      IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &((ip6h)->ip6_src)))) && \
572      (conn_wantpacket_v6((connp), (ira), (ip6h))))
573
574 #define IPCL_CONN_HASH(src, ports, ipst) \
575     (((unsigned)(ntohl((src)) ^ ((ports) >> 24) ^ ((ports) >> 16) ^ \
576      ((ports) >> 8) ^ (ports)) % (ipst)->ips_ipcl_conn_fanout_size)
577
578 #define IPCL_CONN_HASH_V6(src, ports, ipst) \
579     IPCL_CONN_HASH(V4_PART_OF_V6((src)), (ports), (ipst))
580
581 #define IPCL_CONN_MATCH(connp, proto, src, dst, ports) \
582     ((connp)->conn_proto == (proto) && \
583      (connp)->conn_ports == (ports) && \
584      _IPCL_V4_MATCH((connp)->conn_faddr_v6, (src)) && \
585      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (dst)) && \
586      !(connp)->conn_ipv6_v6only)
587
588 #define IPCL_CONN_MATCH_V6(connp, proto, src, dst, ports) \
589     ((connp)->conn_proto == (proto) &&

```

```

590     (connp)->conn_ports == (ports) && \
591     IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(src)) && \
592     IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(dst)))
593
594 #define IPCL_PORT_HASH(port, size) \
595     (((port) >> 8) ^ (port)) & ((size) - 1)
596
597 #define IPCL_BIND_HASH(lport, ipst) \
598     ((unsigned)((lport) >> 8) ^ (lport)) % \
599     (ipst)->ips_ipcl_bind_fanout_size)
600
601 #define IPCL_BIND_MATCH(connp, proto, laddr, lport) \
602     ((connp)->conn_proto == (proto) && \
603      (connp)->conn_lport == (lport) && \
604      (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
605      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))) && \
606      !(connp)->conn_ipv6_v6only)
607
608 #define IPCL_BIND_MATCH_V6(connp, proto, laddr, lport) \
609     ((connp)->conn_proto == (proto) && \
610      (connp)->conn_lport == (lport) && \
611      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) || \
612      IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6)))
613
614 /*
615  * We compare conn_laddr since it captures both connected and a bind to
616  * a multicast or broadcast address.
617  * The caller needs to match the zoneid and also call conn_wantpacket
618  * for multicast, broadcast, or when conn_incoming_ifindex is set.
619  */
620 #define IPCL_UDP_MATCH(connp, lport, laddr, fport, faddr) \
621     (((connp)->conn_lport == (lport)) && \
622      (( _IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
623       (_IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) && \
624       (_IPCL_V4_MATCH_ANY((connp)->conn_faddr_v6) || \
625       (_IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)) && \
626       (connp)->conn_fport == (fport)))))) && \
627      !(connp)->conn_ipv6_v6only)
628
629 /*
630  * We compare conn_laddr since it captures both connected and a bind to
631  * a multicast or broadcast address.
632  * The caller needs to match the zoneid and also call conn_wantpacket_v6
633  * for multicast or when conn_incoming_ifindex is set.
634  */
635 #define IPCL_UDP_MATCH_V6(connp, lport, laddr, fport, faddr) \
636     (((connp)->conn_lport == (lport)) && \
637      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
638      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) && \
639      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) || \
640      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(faddr)) && \
641      (connp)->conn_fport == (fport))))))
642
643 #define IPCL_IPTUN_HASH(laddr, faddr) \
644     ((ntohl(laddr) ^ (ntohl(faddr) << 24) | (ntohl(faddr) >> 8))) % \
645     ipcl iptun_fanout_size)
646
647 #define IPCL_IPTUN_HASH_V6(laddr, faddr) \
648     IPCL_IPTUN_HASH((laddr)->s6_addr32[0] ^ (laddr)->s6_addr32[1] ^ \
649     (faddr)->s6_addr32[2] ^ (faddr)->s6_addr32[3], \
650     (faddr)->s6_addr32[0] ^ (faddr)->s6_addr32[1] ^ \
651     (laddr)->s6_addr32[2] ^ (laddr)->s6_addr32[3])
652
653 #define IPCL_IPTUN_MATCH(connp, laddr, faddr) \
654     (_IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) && \
655     _IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)))

```

```

657 #define IPCL_IPTUN_MATCH_V6(connp, laddr, faddr) \
658     (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, (laddr)) && \
659     IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, (faddr)))

661 #define IPCL_UDP_HASH(lport, ipst) \
662     IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_udp_fanout_size)

664 #define IPCL_DCCP_HASH(lport, ipst) \
665     IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_dccp_fanout_size)

667 #endif /* ! codereview */
668 #define CONN_G_HASH_SIZE      1024

670 /* Raw socket hash function. */
671 #define IPCL_RAW_HASH(lport, ipst) \
672     IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_raw_fanout_size)

674 /*
675  * This is similar to IPCL_BIND_MATCH except that the local port check
676  * is changed to a wildcard port check.
677  * We compare conn_laddr since it captures both connected and a bind to
678  * a multicast or broadcast address.
679  */
680 #define IPCL_RAW_MATCH(connp, proto, laddr) \
681     ((connp)->conn_proto == (proto) && \
682     (connp)->conn_lport == 0 && \
683     (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
684     _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))))

686 #define IPCL_RAW_MATCH_V6(connp, proto, laddr) \
687     ((connp)->conn_proto == (proto) && \
688     (connp)->conn_lport == 0 && \
689     (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
690     IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr))))

692 /* Function prototypes */
693 extern void ipcl_g_init(void);
694 extern void ipcl_init(ip_stack_t *);
695 extern void ipcl_g_destroy(void);
696 extern void ipcl_destroy(ip_stack_t *);
697 extern conn_t *ipcl_conn_create(uint32_t, int, netstack_t *);
698 extern void ipcl_conn_destroy(conn_t *);

700 void ipcl_hash_insert_wildcard(connf_t *, conn_t *);
701 void ipcl_hash_remove(conn_t *);
702 void ipcl_hash_remove_locked(conn_t *connp, connf_t *connfp);

704 extern int ipcl_bind_insert(conn_t *);
705 extern int ipcl_bind_insert_v4(conn_t *);
706 extern int ipcl_bind_insert_v6(conn_t *);
707 extern int ipcl_conn_insert(conn_t *);
708 extern int ipcl_conn_insert_v4(conn_t *);
709 extern int ipcl_conn_insert_v6(conn_t *);
710 extern conn_t *ipcl_get_next_conn(connf_t *, conn_t *, uint32_t);

712 conn_t *ipcl_classify_v4(mblk_t *, uint8_t, uint_t, ip_rcv_attr_t *,
713     ip_stack_t *);
714 conn_t *ipcl_classify_v6(mblk_t *, uint8_t, uint_t, ip_rcv_attr_t *,
715     ip_stack_t *);
716 conn_t *ipcl_classify(mblk_t *, ip_rcv_attr_t *, ip_stack_t *);
717 conn_t *ipcl_classify_raw(mblk_t *, uint8_t, uint32_t, ipha_t *,
718     ip6_t *, ip_rcv_attr_t *, ip_stack_t *);
719 conn_t *ipcl iptun_classify_v4(ipaddr_t *, ipaddr_t *, ip_stack_t *);
720 conn_t *ipcl iptun_classify_v6(in6_addr_t *, in6_addr_t *, ip_stack_t *);
721 void ipcl_globalhash_insert(conn_t *);

```

```

722 void ipcl_globalhash_remove(conn_t *);
723 void ipcl_walk(pfv_t, void *, ip_stack_t *);
724 conn_t *ipcl_tcp_lookup_reversed_ipv4(ipha_t *, tcpha_t *, int, ip_stack_t *);
725 conn_t *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpha_t *, int, uint_t,
726     ip_stack_t *);
727 conn_t *ipcl_lookup_listener_v4(uint16_t, ipaddr_t, zoneid_t, ip_stack_t *);
728 conn_t *ipcl_lookup_listener_v6(uint16_t, in6_addr_t *, uint_t, zoneid_t,
729     ip_stack_t *);
730 int conn_trace_ref(conn_t *);
731 int conn_untrace_ref(conn_t *);
732 void ipcl_conn_cleanup(conn_t *);
733 extern uint_t conn_rcvancillary_size(conn_t *, crb_t, ip_rcv_attr_t *,
734     mblk_t *, ip_pkt_t *);
735 extern void conn_rcvancillary_add(conn_t *, crb_t, ip_rcv_attr_t *,
736     ip_pkt_t *, uchar_t *, uint_t);
737 conn_t *ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *, ipha_t *, tcpha_t *,
738     ip_stack_t *);
739 conn_t *ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *, ip6_t *, tcpha_t *,
740     ip_stack_t *);

742 extern int ip_create_helper_stream(conn_t *, ldi_ident_t);
743 extern void ip_free_helper_stream(conn_t *);
744 extern int ip_helper_stream_setup(queue_t *, dev_t *, int, int,
745     cred_t *, boolean_t);

747 #ifdef __cplusplus
748 }
749 #endif

751 #endif /* _INET_IPCLASSIFIER_H */

```


new/usr/src/uts/common/inet/mib2.h

1

```
*****
61137 Wed Jun 13 12:05:11 2012
new/usr/src/uts/common/inet/mib2.h
%B
*****
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 *
21 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
22 */
23 /* Copyright (c) 1990 Mentat Inc. */

25 #ifndef _INET_MIB2_H
26 #define _INET_MIB2_H

28 #include <netinet/in.h> /* For in6_addr_t */
29 #include <sys/tsocket/label.h> /* For brange_t */
30 #include <sys/tsocket/label_macro.h> /* For brange_t */

32 #ifdef __cplusplus
33 extern "C" {
34 #endif

36 /*
37  * The IPv6 parts of this are derived from:
38  *   RFC 2465
39  *   RFC 2466
40  *   RFC 2452
41  *   RFC 2454
42  */

44 /*
45  * SNMP set/get via M_PROTO T_OPTMGMT_REQ. Structure is that used
46  * for [gs]tsocket() calls. get uses T_CURRENT, set uses T_NEOGTIATE
47  * MGMT_flags value. The following definition of ophdr is taken from
48  * socket.h:
49  *
50  * An option specification consists of an ophdr, followed by the value of
51  * the option. An options buffer contains one or more options. The len
52  * field of ophdr specifies the length of the option value in bytes. This
53  * length must be a multiple of sizeof(long) (use OPTLEN macro).
54  *
55  * struct ophdr {
56  *     long    level;  protocol level affected
57  *     long    name;   option to modify
58  *     long    len;    length of option value
59  * };
60  *
61  * #define OPTLEN(x) (((x) + sizeof(long) - 1) / sizeof(long)) * sizeof(long))

```

new/usr/src/uts/common/inet/mib2.h

2

```
62 * #define OPTVAL(opt) ((char *)(opt + 1))
63 *
64 * For get requests (T_CURRENT), any MIB2_xxx value can be used (only
65 * "get all" is supported, so all modules get a copy of the request to
66 * return everything it knows. In general, we use MIB2_IP. There is
67 * one exception: in general, IP will not report information related to
68 * ire_testhidden and IRE_IF_CLONE routes (e.g., in the MIB2_IP_ROUTE
69 * table). However, using the special value EXPER_IP_AND_ALL_IRES will cause
70 * all information to be reported. This special value should only be
71 * used by IPMP-aware low-level utilities (e.g. in.mpathd).
72 *
73 * IMPORTANT: some fields are grouped in a different structure than
74 * suggested by MIB-II, e.g., checksum error counts. The original MIB-2
75 * field name has been retained. Field names beginning with "mi" are not
76 * defined in the MIB but contain important & useful information maintained
77 * by the corresponding module.
78 */
79 #ifndef IPPROTO_MAX
80 #define IPPROTO_MAX 256
81 #endif

83 #define MIB2_SYSTEM (IPPROTO_MAX+1)
84 #define MIB2_INTERFACES (IPPROTO_MAX+2)
85 #define MIB2_AT (IPPROTO_MAX+3)
86 #define MIB2_IP (IPPROTO_MAX+4)
87 #define MIB2_ICMP (IPPROTO_MAX+5)
88 #define MIB2_TCP (IPPROTO_MAX+6)
89 #define MIB2_UDP (IPPROTO_MAX+7)
90 #define MIB2_EGP (IPPROTO_MAX+8)
91 #define MIB2_CMOT (IPPROTO_MAX+9)
92 #define MIB2_TRANSMISSION (IPPROTO_MAX+10)
93 #define MIB2_SNMP (IPPROTO_MAX+11)
94 #define MIB2_IP6 (IPPROTO_MAX+12)
95 #define MIB2_ICMP6 (IPPROTO_MAX+13)
96 #define MIB2_TCP6 (IPPROTO_MAX+14)
97 #define MIB2_UDP6 (IPPROTO_MAX+15)
98 #define MIB2_SCTP (IPPROTO_MAX+16)
99 #define MIB2_DCCP (IPPROTO_MAX+17)
100 #endif /* ! codereview */

102 /*
103  * Define range of levels for use with MIB2_*
104  */
105 #define MIB2_RANGE_START (IPPROTO_MAX+1)
106 #define MIB2_RANGE_END (IPPROTO_MAX+17)
99 #define MIB2_RANGE_END (IPPROTO_MAX+16)

109 #define EXPER 1024 /* experimental - not part of mib */
110 #define EXPER_IGMP (EXPER+1)
111 #define EXPER_DVMRP (EXPER+2)
112 #define EXPER_RAWIP (EXPER+3)
113 #define EXPER_IP_AND_ALL_IRES (EXPER+4)

115 /*
116  * Define range of levels for experimental use
117  */
118 #define EXPER_RANGE_START (EXPER+1)
119 #define EXPER_RANGE_END (EXPER+4)

121 #define BUMP_MIB(s, x) {
122     extern void __dtrace_probe__mib_##x(int, void *);
123     void *stataddr = &((s)->x);
124     __dtrace_probe__mib_##x(1, stataddr);
125     (s)->x++;
126 }
unchanged_portion_omitted

```

```
1787 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1788 #pragma pack()
1789 #endif

1791 /*
1792  * the DCCP group
1793  */
1794 #define MIB2_DCCP_CONN          18

1796 #define MIB2_DCCP_closed        1
1797 #define MIB2_DCCP_listen        2

1799 /* Pack data to make struct size the same for 32- and 64-bits */
1800 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1801 #pragma pack(4)
1802 #endif

1804 typedef struct mib2_dccp {
1806     Counter dccpActiveOpens;
1808     int      dccpEntrySize;
1809 } mib2_dccp_t;

1811 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1812 #pragma pack()
1813 #endif

1815 /* Pack data to make struct size the same for 32- and 64-bits */
1816 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1817 #pragma pack(4)
1818 #endif

1820 typedef struct mib2_dccpConnEntry {
1821     int      dccpConnState;
1822     IpAddress dccpConnLocalAddress;
1823     int      dccpConnLocalPort;
1824     IpAddress dccpConnRemAddress;
1825     int      dccpConnRemPort;

1827     uint32_t dccpConnCreationProcess;
1828     uint64_t dccpConnCreationTime;
1829 } mib2_dccpConnEntry_t;
1830 #define MIB_FIRST_NEW_ELM_mib2_dccpConnEntry_t dccpConnCreationProcess

1832 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1833 #pragma pack()
1834 #endif
1835 #endif /* ! codereview */

1837 #ifdef __cplusplus
1838 }
1839 #endif

1841 #endif /* _INET_MIB2_H */
```

```

*****
46352 Wed Jun 13 12:05:13 2012
new/usr/src/uts/common/inet/sctp/sctp_impl.h
%B
*****
_____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 beginning (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_rtt;        /* 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_SUPP_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;     /* rcv pkts */
731     uint64_t    sctp_ibchunks;  /* rcv control chunks */
732     uint64_t    sctp_idchunks;  /* rcv ordered data chunks */
733     uint64_t    sctp_iudchunks; /* rcv 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_rcv_adaptation : 1,  /* rcv 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_flowctrlrd : 1,    /* upper layer flow controlled */
809         sctp_dummy : 5;

810     } sctp_bits;
811     struct {
812         uint32_t

814         sctp_recvsndrcvinfo : 1,
815         sctp_recvassocevt : 1,
816         sctp_recvpathevt : 1,

```

```

817         sctp_rcvsendfailevnt : 1,
819         sctp_rcvpeererr : 1,
820         sctp_rcvshutdownevnt : 1,
821         sctp_rcvpevnt : 1,
822         sctp_rcvvalvnt : 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_rcv_adaptation sctp_bits.sctp_rcv_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_flowctrl sctp_bits.sctp_flowctrl

849 #define sctp_rcvsnrdrcvinfo sctp_events.sctp_rcvsnrdrcvinfo
850 #define sctp_rcvassocevt sctp_events.sctp_rcvassocevt
851 #define sctp_rcvpathevt sctp_events.sctp_rcvpathevt
852 #define sctp_rcvsendfailevnt sctp_events.sctp_rcvsendfailevnt
853 #define sctp_rcvpeererr sctp_events.sctp_rcvpeererr
854 #define sctp_rcvshutdownevnt sctp_events.sctp_rcvshutdownevnt
855 #define sctp_rcvpevnt sctp_events.sctp_rcvpevnt
856 #define sctp_rcvvalvnt sctp_events.sctp_rcvvalvnt

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_rcvq_lock;
901 mblk_t      *sctp_rcvq;
902 mblk_t      *sctp_rcvq_tail;
903 taskq_t     sctp_rcvq_tq;

905 /* IPv6 ancillary data */
906 uint_t      sctp_rcvifindex; /* last rcvd IPV6_RCVPKTINFO */
907 uint_t      sctp_rcvhops; /* " IPV6_RECVMHOPLIMIT */
908 uint_t      sctp_rcvclass; /* " IPV6_RECVMCLASS */
909 ip6_hbh_t   *sctp_hopopts; /* " IPV6_RECVMHOPOPTS */
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_rthdrhlen;

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 retransmitted */
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; /* rcv control chunks */
946 uint64_t    sctp_cum_idchunks; /* rcv ordered data chunks */
947 uint64_t    sctp_cum_iudchunks; /* rcv 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 void      *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_oob_shutdown_ack(mblk_t *, uint_t, ip_rcv_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_rcv_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_rcv_attr_t *);
1124 extern void       sctp_send_abort(sctp_t *, uint32_t, uint16_t, char *, size_t,
1125                          mblk_t *, int, boolean_t, ip_rcv_attr_t *);
1126 extern void       sctp_oob_send_abort(uint32_t, uint16_t, char *, size_t,
1127                          const mblk_t *, int, boolean_t, ip_rcv_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_rcv_attr_t *);
1132 extern void       sctp_send_initack(sctp_t *, sctp_hdr_t *, sctp_chunk_hdr_t *,
1133                          mblk_t *, ip_rcv_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_iphlen(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 */

```


new/usr/src/uts/common/inet/tcp/tcp.c

1

```
*****
129612 Wed Jun 13 12:05:15 2012
new/usr/src/uts/common/inet/tcp/tcp.c
%B
*****
_____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     queue_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);

```

new/usr/src/uts/common/inet/tcp/tcp.c

2

```

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_cpuid = 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_cpuid = connp->conn_cpuid;

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_rcv_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 Wed Jun 13 12:05:20 2012
new/usr/src/uts/common/inet/tcp/tcp_opt_data.c
%B
*****
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 */
21 /*
22 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2011 Nexenta Systems, Inc. All rights reserved.
24 */

26 #include <sys/types.h>
27 #include <sys/stream.h>
28 #define _SUN_TPI_VERSION 2
29 #include <sys/tihdr.h>
30 #include <sys/socket.h>
31 #include <sys/xti_xtiopt.h>
32 #include <sys/xti_inet.h>
33 #include <sys/policy.h>

35 #include <inet/common.h>
36 #include <netinet/ip6.h>
37 #include <netinet/ip.h>

39 #include <netinet/in.h>
40 #include <netinet/tcp.h>
41 #include <inet/optcom.h>
42 #include <inet/proto_set.h>
43 #include <inet/tcp_impl.h>

45 static int tcp_opt_default(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);

47 #endif /* ! codereview */
48 /*
49  * Table of all known options handled on a TCP protocol stack.
50  *
51  * Note: This table contains options processed by both TCP and IP levels
52  * and is the superset of options that can be performed on a TCP over IP
53  * stack.
54  */
55 opdes_t tcp_opt_arr[] = {

57 { SO_LINGER, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0,
58   sizeof (struct linger), 0 },

60 { SO_DEBUG, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
61 { SO_KEEPA_LIVE, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

```

```

62 { SO_DONTROUTE, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
63 { SO_USELOOPBACK, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
64 },
65 { SO_BROADCAST, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
66 { SO_REUSEADDR, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
67 { SO_OOBINLINE, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
68 { SO_TYPE, SOL_SOCKET, OA_R, OA_R, OP_NP, 0, sizeof (int), 0 },
69 { SO_SNDBUF, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
70 { SO_RCVBUF, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
71 { SO_SNDTIMEO, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0,
72   sizeof (struct timeval), 0 },
73 { SO_RCVTIMEO, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0,
74   sizeof (struct timeval), 0 },
75 { SO_DGRAM_ERRIND, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
76 },
77 { SO_SND_COPYAVOID, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
78 { SO_ANON_MLP, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
79   0 },
80 { SO_MAC_EXEMPT, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
81   0 },
82 { SO_MAC_IMPLICIT, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
83   0 },
84 { SO_ALLZONES, SOL_SOCKET, OA_R, OA_RW, OP_CONFIG, 0, sizeof (int),
85   0 },
86 { SO_EXCLBIND, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

88 { SO_DOMAIN, SOL_SOCKET, OA_R, OA_R, OP_NP, 0, sizeof (int), 0 },

90 { SO_PROTOTYPE, SOL_SOCKET, OA_R, OA_R, OP_NP, 0, sizeof (int), 0 },

92 { TCP_NODELAY, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
93 },
94 { TCP_MAXSEG, IPPROTO_TCP, OA_R, OA_R, OP_NP, 0, sizeof (uint_t),
95   536 },

97 { TCP_NOTIFY_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
98   OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

100 { TCP_ABORT_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
101   OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

103 { TCP_CONN_NOTIFY_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
104   OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

106 { TCP_CONN_ABORT_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP,
107   OP_DEF_FN, sizeof (int), -1 /* not initialized */ },

109 { TCP_RECVDSTADDR, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int),
110   0 },

112 { TCP_ANONPRIVBIND, IPPROTO_TCP, OA_R, OA_RW, OP_PRIVPORT, 0,
113   sizeof (int), 0 },

115 { TCP_EXCLBIND, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
116 },

118 { TCP_INIT_CWND, IPPROTO_TCP, OA_RW, OA_RW, OP_CONFIG, 0,
119   sizeof (int), 0 },

121 { TCP_KEEPA_LIVE_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0,
122   sizeof (int), 0 },

124 { TCP_KEEPI_DLE, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

126 { TCP_KEEPCNT, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },

```

```

128 { TCP_KEEPIPTVL, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
130 { TCP_KEEPLIVE_ABORT_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0,
131     sizeof (int), 0 },
133 { TCP_CORK, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
135 { TCP_RTO_INITIAL, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (uint32_t), 0 },
137 { TCP_RTO_MIN, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (uint32_t), 0 },
139 { TCP_RTO_MAX, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (uint32_t), 0 },
141 { TCP_LINGER2, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
143 { IP_OPTIONS,    IPPROTO_IP, OA_RW, OA_RW, OP_NP,
144     (OP_VARLEN|OP_NODEFAULT),
145     IP_MAX_OPT_LENGTH + IP_ADDR_LEN, -1 /* not initialized */ },
146 { T_IP_OPTIONS,  IPPROTO_IP, OA_RW, OA_RW, OP_NP,
147     (OP_VARLEN|OP_NODEFAULT),
148     IP_MAX_OPT_LENGTH + IP_ADDR_LEN, -1 /* not initialized */ },
150 { IP_TOS,        IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
151 { T_IP_TOS,      IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
152 { IP_TTL,        IPPROTO_IP, OA_RW, OA_RW, OP_NP, OP_DEF_FN,
153     sizeof (int), -1 /* not initialized */ },
155 { IP_SEC_OPT,   IPPROTO_IP, OA_RW, OA_RW, OP_NP, OP_NODEFAULT,
156     sizeof (ipsec_req_t), -1 /* not initialized */ },
158 { IP_BOUND_IF,  IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0,
159     sizeof (int), 0 /* no ifindex */ },
161 { IP_UNSPEC_SRC, IPPROTO_IP, OA_R, OA_RW, OP_RAW, 0,
162     sizeof (int), 0 },
164 { IPV6_UNICAST_HOPS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, OP_DEF_FN,
165     sizeof (int), -1 /* not initialized */ },
167 { IPV6_BOUND_IF,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
168     sizeof (int), 0 /* no ifindex */ },
170 { IP_DONTFRAG,   IPPROTO_IP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
172 { IP_NEXTHOP,    IPPROTO_IP, OA_R, OA_RW, OP_CONFIG, 0,
173     sizeof (in_addr_t), -1 /* not initialized */ },
175 { IPV6_UNSPEC_SRC, IPPROTO_IPV6, OA_R, OA_RW, OP_RAW, 0,
176     sizeof (int), 0 },
178 { IPV6_PKTINFO,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
179     (OP_NODEFAULT|OP_VARLEN),
180     sizeof (struct in6_pktinfo), -1 /* not initialized */ },
181 { IPV6_NEXTHOP,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
182     OP_NODEFAULT,
183     sizeof (sin6_t), -1 /* not initialized */ },
184 { IPV6_HOPOPTS,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
185     (OP_VARLEN|OP_NODEFAULT), 255*8,
186     -1 /* not initialized */ },
187 { IPV6_DSTOPTS,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
188     (OP_VARLEN|OP_NODEFAULT), 255*8,
189     -1 /* not initialized */ },
190 { IPV6_RTHDRDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
191     (OP_VARLEN|OP_NODEFAULT), 255*8,
192     -1 /* not initialized */ },
193 { IPV6_RTHDR,    IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,

```

```

194     (OP_VARLEN|OP_NODEFAULT), 255*8,
195     -1 /* not initialized */ },
196 { IPV6_TCLASS,   IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
197     OP_NODEFAULT,
198     sizeof (int), -1 /* not initialized */ },
199 { IPV6_PATHMTU,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP,
200     OP_NODEFAULT,
201     sizeof (struct ip6_mtuinfo), -1 /* not initialized */ },
202 { IPV6_DONTFRAG, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
203     sizeof (int), 0 },
204 { IPV6_USE_MIN_MTU, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
205     sizeof (int), 0 },
206 { IPV6_V6ONLY,   IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
207     sizeof (int), 0 },
209 /* Enable receipt of ancillary data */
210 { IPV6_RECVPKTINFO, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
211     sizeof (int), 0 },
212 { IPV6_RECVHOPLIMIT, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
213     sizeof (int), 0 },
214 { IPV6_RECVHOPOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
215     sizeof (int), 0 },
216 { _OLD_IPV6_RECVDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
217     sizeof (int), 0 },
218 { IPV6_RECVDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
219     sizeof (int), 0 },
220 { IPV6_RECVRTHDR,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
221     sizeof (int), 0 },
222 { IPV6_RECVRTHDRDSTOPTS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
223     sizeof (int), 0 },
224 { IPV6_RECVTCLASS, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
225     sizeof (int), 0 },
227 { IPV6_SEC_OPT,  IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, OP_NODEFAULT,
228     sizeof (ipsec_req_t), -1 /* not initialized */ },
229 { IPV6_SRC_PREFERENCES, IPPROTO_IPV6, OA_RW, OA_RW, OP_NP, 0,
230     sizeof (uint32_t), IPV6_PREFER_SRC_DEFAULT },
231 };
233 /*
234  * Table of all supported levels
235  * Note: Some levels (e.g. XTI_GENERIC) may be valid but may not have
236  * any supported options so we need this info separately.
237  *
238  * This is needed only for topmost tpi providers and is used only by
239  * XTI interfaces.
240  */
241 optlevel_t    tcp_valid_levels_arr[] = {
242     XTI_GENERIC,
243     SOL_SOCKET,
244     IPPROTO_TCP,
245     IPPROTO_IP,
246     IPPROTO_IPV6
247 };
250 #define TCP_OPT_ARR_CNT        A_CNT(tcp_opt_arr)
251 #define TCP_VALID_LEVELS_CNT  A_CNT(tcp_valid_levels_arr)
253 uint_t tcp_max_optsize; /* initialized when TCP driver is loaded */
255 /*
256  * Initialize option database object for TCP
257  *
258  * This object represents database of options to search passed to
259  * {sock,tpi}optcom_req() interface routine to take care of option

```

```

260 * management and associated methods.
261 */

263 optdb_obj_t tcp_opt_obj = {
264     tcp_opt_default,      /* TCP default value function pointer */
265     tcp_tpi_opt_get,     /* TCP get function pointer */
266     tcp_tpi_opt_set,     /* TCP set function pointer */
267     TCP_OPT_ARR_CNT,    /* TCP option database count of entries */
268     tcp_opt_arr,        /* TCP option database */
269     TCP_VALID_LEVELS_CNT, /* TCP valid level count of entries */
270     tcp_valid_levels_arr /* TCP valid level array */
271 };

273 /* Maximum TCP initial cwin (start/restart). */
274 #define TCP_MAX_INIT_CWND    16

276 static int tcp_max_init_cwnd = TCP_MAX_INIT_CWND;

278 /*
279  * Some TCP options can be "set" by requesting them in the option
280  * buffer. This is needed for XTI feature test though we do not
281  * allow it in general. We interpret that this mechanism is more
282  * applicable to OSI protocols and need not be allowed in general.
283  * This routine filters out options for which it is not allowed (most)
284  * and lets through those (few) for which it is. [ The XTI interface
285  * test suite specifics will imply that any XTI_GENERIC level XTI_* if
286  * ever implemented will have to be allowed here ].
287  */
288 static boolean_t
289 tcp_allow_connopt_set(int level, int name)
290 {
291     switch (level) {
292     case IPPROTO_TCP:
293         switch (name) {
294             case TCP_NODELAY:
295                 return (B_TRUE);
296             default:
297                 return (B_FALSE);
298         }
299     }
300     /*NOTREACHED*/
301     default:
302         return (B_FALSE);
303     }
304     /*NOTREACHED*/
305 }

307 /*
308  * This routine gets default values of certain options whose default
309  * values are maintained by protocol specific code
310  */
311 /* ARGSUSED */
312 int
313 tcp_opt_default(queue_t *q, int level, int name, uchar_t *ptr)
314 {
315     int32_t *i1 = (int32_t *)ptr;
316     tcp_stack_t *tcps = Q_TO_TCP(q)->tcp_tcps;

318     switch (level) {
319     case IPPROTO_TCP:
320         switch (name) {
321             case TCP_NOTIFY_THRESHOLD:
322                 *i1 = tcps->tcps_ip_notify_interval;
323                 break;
324             case TCP_ABORT_THRESHOLD:
325                 *i1 = tcps->tcps_ip_abort_interval;

```

```

326         break;
327     case TCP_CONN_NOTIFY_THRESHOLD:
328         *i1 = tcps->tcps_ip_notify_cinterval;
329         break;
330     case TCP_CONN_ABORT_THRESHOLD:
331         *i1 = tcps->tcps_ip_abort_cinterval;
332         break;
333     default:
334         return (-1);
335     }
336     break;
337 case IPPROTO_IP:
338     switch (name) {
339     case IP_TTL:
340         *i1 = tcps->tcps_ipv4_ttl;
341         break;
342     default:
343         return (-1);
344     }
345     break;
346 case IPPROTO_IPV6:
347     switch (name) {
348     case IPV6_UNICAST_HOPS:
349         *i1 = tcps->tcps_ipv6_hoplimit;
350         break;
351     default:
352         return (-1);
353     }
354     break;
355 default:
356     return (-1);
357 }
358 return (sizeof (int));
359 }

361 /*
362  * TCP routine to get the values of options.
363  */
364 int
365 tcp_opt_get(conn_t *connp, int level, int name, uchar_t *ptr)
366 {
367     int *i1 = (int *)ptr;
368     tcp_t *tcp = connp->conn_tcp;
369     conn_opt_arg_t coas;
370     int retval;

372     coas.coa_connp = connp;
373     coas.coa_ixa = connp->conn_ixa;
374     coas.coa_ipp = &connp->conn_xmit_ipp;
375     coas.coa_ancillary = B_FALSE;
376     coas.coa_changed = 0;

378     switch (level) {
379     case SOL_SOCKET:
380         switch (name) {
381             case SO_SND_COPYAVOID:
382                 *i1 = tcp->tcp_snd_zcopy_on ?
383                     SO_SND_COPYAVOID : 0;
384                 return (sizeof (int));
385             case SO_ACCEPTCONN:
386                 *i1 = (tcp->tcp_state == TCPS_LISTEN);
387                 return (sizeof (int));
388         }
389         break;
390     case IPPROTO_TCP:
391         switch (name) {

```

```

392     case TCP_NODELAY:
393         *il = (tcp->tcp_naglim == 1) ? TCP_NODELAY : 0;
394         return (sizeof (int));
395     case TCP_MAXSEG:
396         *il = tcp->tcp_mss;
397         return (sizeof (int));
398     case TCP_NOTIFY_THRESHOLD:
399         *il = (int)tcp->tcp_first_timer_threshold;
400         return (sizeof (int));
401     case TCP_ABORT_THRESHOLD:
402         *il = tcp->tcp_second_timer_threshold;
403         return (sizeof (int));
404     case TCP_CONN_NOTIFY_THRESHOLD:
405         *il = tcp->tcp_first_ctimer_threshold;
406         return (sizeof (int));
407     case TCP_CONN_ABORT_THRESHOLD:
408         *il = tcp->tcp_second_ctimer_threshold;
409         return (sizeof (int));
410     case TCP_INIT_CWND:
411         *il = tcp->tcp_init_cwnd;
412         return (sizeof (int));
413     case TCP_KEEPA_LIVE_THRESHOLD:
414         *il = tcp->tcp_ka_interval;
415         return (sizeof (int));
417     /*
418      * TCP_KEEPI_DLE expects value in seconds, but
419      * tcp_ka_interval is in milliseconds.
420      */
421     case TCP_KEEPI_DLE:
422         *il = tcp->tcp_ka_interval / 1000;
423         return (sizeof (int));
424     case TCP_KEEPCNT:
425         *il = tcp->tcp_ka_cnt;
426         return (sizeof (int));
428     /*
429      * TCP_KEEPI_NTVL expects value in seconds, but
430      * tcp_ka_rinterval is in milliseconds.
431      */
432     case TCP_KEEPI_NTVL:
433         *il = tcp->tcp_ka_rinterval / 1000;
434         return (sizeof (int));
435     case TCP_KEEPA_LIVE_ABORT_THRESHOLD:
436         *il = tcp->tcp_ka_abort_thres;
437         return (sizeof (int));
438     case TCP_CORK:
439         *il = tcp->tcp_cork;
440         return (sizeof (int));
441     case TCP_RTO_INITIAL:
442         *il = tcp->tcp_rto_initial;
443         return (sizeof (uint32_t));
444     case TCP_RTO_MIN:
445         *il = tcp->tcp_rto_min;
446         return (sizeof (uint32_t));
447     case TCP_RTO_MAX:
448         *il = tcp->tcp_rto_max;
449         return (sizeof (uint32_t));
450     case TCP_LINGER2:
451         *il = 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 *il = (int *)invalp;
502     boolean_t onoff = (*il == 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 (*il > tcps->tcps_max_buf) {
606             *outlenp = 0;
607             return (ENOBUFS);
608         }
609         if (checkonly)
610             break;
612         connp->conn_sndbuf = *il;
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 (*il > tcps->tcps_max_buf) {
635             *outlenp = 0;
636             return (ENOBUFS);
637         }
638         /* Silently ignore zero */
639         if (!checkonly && *il != 0) {
640             *il = MSS_ROUNDUP(*il, tcp->tcp_mss);
641             (void) tcp_rwnd_set(tcp, *il);
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 = *il ? 1 : tcp->tcp_mss;
667         break;
668     case TCP_NOTIFY_THRESHOLD:
669         if (!checkonly)
670             tcp->tcp_first_timer_threshold = *il;
671         break;
672     case TCP_ABORT_THRESHOLD:
673         if (!checkonly)
674             tcp->tcp_second_timer_threshold = *il;
675         break;
676     case TCP_CONN_NOTIFY_THRESHOLD:
677         if (!checkonly)
678             tcp->tcp_first_ctimer_threshold = *il;
679         break;
680     case TCP_CONN_ABORT_THRESHOLD:
681         if (!checkonly)
682             tcp->tcp_second_ctimer_threshold = *il;
683         break;
684     case TCP_RECVSTADDR:
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;
694
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     }
711
712     tcp->tcp_init_cwnd = val;
713
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;
724
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         *il *= 1000;
734         /* FALLTHRU */
735
736     case TCP_KEEPALIVE_THRESHOLD:
737         if (checkonly)
738             break;
739
740         if (*il < tcps->tcps_keepalive_interval_low ||
741             *il > tcps->tcps_keepalive_interval_high) {
742             *outlenp = 0;
743             return (EINVAL);
744         }
745         if (*il != tcp->tcp_ka_interval) {
746             tcp->tcp_ka_interval = *il;
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;
762
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;
772
773         if (*il == 0) {
774             return (EINVAL);
775         }
776         else if (tcp->tcp_ka_rinterval == 0) {
777             if ((tcp->tcp_ka_abort_thres / *il) <
778                 tcp->tcp_rto_min ||
779                 (tcp->tcp_ka_abort_thres / *il) >
780                 tcp->tcp_rto_max)
781                 return (EINVAL);
782
783             tcp->tcp_ka_rinterval =
784                 tcp->tcp_ka_abort_thres / *il;
785         }
786         else {
787             if ((*il * tcp->tcp_ka_rinterval) <
788                 tcps->tcps_keepalive_abort_interval_low ||
789                 (*il * tcp->tcp_ka_rinterval) >

```



```

788         tcps->tcps_keepalive_abort_interval_high)
789         return (EINVAL);
790         tcp->tcp_ka_abort_thres =
791         (*il * tcp->tcp_ka_rinterval);
792     }
793     tcp->tcp_ka_cnt = *il;
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 ((*il * 1000) < tcp->tcp_rto_min ||
805         (*il * 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 / (*il * 1000);
811     } else {
812         if ((*il * tcp->tcp_ka_cnt * 1000) <
813             tcps->tcps_keepalive_abort_interval_low ||
814             (*il * tcp->tcp_ka_cnt * 1000) >
815             tcps->tcps_keepalive_abort_interval_high)
816             return (EINVAL);
817         tcp->tcp_ka_abort_thres =
818         (*il * tcp->tcp_ka_cnt * 1000);
819     }
820     tcp->tcp_ka_rinterval = *il * 1000;
821     break;
822 case TCP_KEEPALIVE_ABORT_THRESHOLD:
823     if (!checkonly) {
824         if (*il <
825             tcps->tcps_keepalive_abort_interval_low ||
826             *il >
827             tcps->tcps_keepalive_abort_interval_high) {
828             *outlenp = 0;
829             return (EINVAL);
830         }
831         tcp->tcp_ka_abort_thres = *il;
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 || *il == 0)
933             break;
934
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 (*il < 0 ||
946         tcps->tcps_fin_wait_2_flush_interval_low/SECONDS >
947         *il ||
948         tcps->tcps_fin_wait_2_flush_interval/SECONDS <
949         *il) {
950         *outlenp = 0;
951         return (EINVAL);
952     }
953     tcp->tcp_fin_wait_2_flush_interval = *il * 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;
1009
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 }
1039
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;
1048
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;
1060
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);
1069
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_OOBLINE_CHANGED) {
1088         if (IPCL_IS_NONSTR(connp))
1089             proto_set_rx_oob_opt(connp, onoff);
1090     }
1091     return (0);
1092 }
```

```

*****
31926 Wed Jun 13 12:05:24 2012
new/usr/src/uts/common/inet/tcp/tcp_socket.c
%B
*****
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) 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /* This file contains all TCP kernel socket related functions. */

28 #include <sys/types.h>
29 #include <sys/strlog.h>
30 #include <sys/policy.h>
31 #include <sys/sockio.h>
32 #include <sys/strsubr.h>
33 #include <sys/strsun.h>
34 #include <sys/queue_impl.h>
35 #include <sys/queue.h>
36 #define _SUN_TPI_VERSION 2
37 #include <sys/tihdr.h>
38 #include <sys/timod.h>
39 #include <sys/tpicommon.h>
40 #include <sys/socketvar.h>

42 #include <inet/common.h>
43 #include <inet/proto_set.h>
44 #include <inet/ip.h>
45 #include <inet/tcp.h>
46 #include <inet/tcp_impl.h>

48 static void      tcp_activate(sock_lower_handle_t, sock_upper_handle_t,
49                             sock_upcalls_t *, int, cred_t *);
50 static int       tcp_accept(sock_lower_handle_t, sock_lower_handle_t,
51                             sock_upper_handle_t, cred_t *);
52 static int       tcp_bind(sock_lower_handle_t, struct sockaddr *,
53                             socklen_t, cred_t *);
54 static int       tcp_listen(sock_lower_handle_t, int, cred_t *);
55 static int       tcp_connect(sock_lower_handle_t, const struct sockaddr *,
56                             socklen_t, sock_connid_t *, cred_t *);
57 static int       tcp_getpeername(sock_lower_handle_t, struct sockaddr *,
58                             socklen_t *, cred_t *);
59 static int       tcp_getsockname(sock_lower_handle_t, struct sockaddr *,
60                             socklen_t *, cred_t *);
61 #endif /* ! codereview */

```

```

62 static int       tcp_getsockopt(sock_lower_handle_t, int, int, void *,
63                             socklen_t *, cred_t *);
64 static int       tcp_setsockopt(sock_lower_handle_t, int, int, const void *,
65                             socklen_t, cred_t *);
66 static int       tcp_sendmsg(sock_lower_handle_t, mblk_t *, struct nmsg_hdr *,
67                             cred_t *);
68 static int       tcp_shutdown(sock_lower_handle_t, int, cred_t *);
69 static void      tcp_clr_flowctrl(sock_lower_handle_t);
70 static int       tcp_ioctl(sock_lower_handle_t, int, intptr_t, int, int32_t *,
71                             cred_t *);
72 static int       tcp_close(sock_lower_handle_t, int, cred_t *);

74 sock_downcalls_t sock_tcp_downcalls = {
75     tcp_activate,
76     tcp_accept,
77     tcp_bind,
78     tcp_listen,
79     tcp_connect,
80     tcp_getpeername,
81     tcp_getsockname,
82     tcp_getsockopt,
83     tcp_setsockopt,
84     tcp_sendmsg,
85     NULL,
86     NULL,
87     NULL,
88     tcp_shutdown,
89     tcp_clr_flowctrl,
90     tcp_ioctl,
91     tcp_close,
92 };

    unchanged_portion_omitted

753 /* ARGSUSED */
754 sock_lower_handle_t
755 tcp_create(int family, int type, int proto, sock_downcalls_t **sock_downcalls,
756           uint_t *smodep, int *errorp, int flags, cred_t *credp)
757 {
758     conn_t          *connp;
759     boolean_t       isv6 = family == AF_INET6;

761 #endif /* ! codereview */
762     if (type != SOCK_STREAM || (family != AF_INET && family != AF_INET6) ||
763         (proto != 0 && proto != IPPROTO_TCP)) {
764         *errorp = EPROTONOSUPPORT;
765         return (NULL);
766     }

768     connp = tcp_create_common(credp, isv6, B_TRUE, errorp);
769     if (connp == NULL) {
770         return (NULL);
771     }

773     /*
774      * Put the ref for TCP. Ref for IP was already put
775      * by ipcl_conn_create. Also make the conn_t globally
776      * visible to walkers.
777      * by ipcl_conn_create. Also Make the conn_t globally
778      * visible to walkers
779      */
778     mutex_enter(&connp->conn_lock);
779     CONN_INC_REF_LOCKED(connp);
780     ASSERT(connp->conn_ref == 2);
781     connp->conn_state_flags &= ~CONN_INCIPIENT;

```

new/usr/src/uts/common/inet/tcp/tcp_socket.c

3

```
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 Wed Jun 13 12:05:25 2012
new/usr/src/uts/common/inet/tcp_impl.h
%B
*****
_____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     (mbk_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     (mbk_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_cur_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 tcps_propinfo_tbl[20].prop_max_uval
444 #define tcps_rexmit_interval_initial tcps_propinfo_tbl[20].prop_cur_uval
445 #define tcps_rexmit_interval_initial_low tcps_propinfo_tbl[20].prop_min_uval
446 #define tcps_rexmit_interval_max_high tcps_propinfo_tbl[21].prop_max_uval
447 #define tcps_rexmit_interval_max tcps_propinfo_tbl[21].prop_cur_uval
448 #define tcps_rexmit_interval_max_low tcps_propinfo_tbl[21].prop_min_uval
449 #define tcps_rexmit_interval_min_high tcps_propinfo_tbl[22].prop_max_uval
450 #define tcps_rexmit_interval_min tcps_propinfo_tbl[22].prop_cur_uval
451 #define tcps_rexmit_interval_min_low tcps_propinfo_tbl[22].prop_min_uval
452 #define tcps_deferred_ack_interval tcps_propinfo_tbl[23].prop_cur_uval
453 #define tcps_snd_lowat_fraction tcps_propinfo_tbl[24].prop_cur_uval
454 #define tcps_dupack_fast_retransmit tcps_propinfo_tbl[25].prop_cur_uval
455 #define tcps_ignore_path_mtu tcps_propinfo_tbl[26].prop_cur_bval
456 #define tcps_smallest_anon_port tcps_propinfo_tbl[27].prop_cur_uval
457 #define tcps_largest_anon_port tcps_propinfo_tbl[28].prop_cur_uval
458 #define tcps_xmit_hiwat tcps_propinfo_tbl[29].prop_cur_uval
459 #define tcps_xmit_lowat tcps_propinfo_tbl[30].prop_cur_uval
460 #define tcps_recv_hiwat tcps_propinfo_tbl[31].prop_cur_uval
461 #define tcps_recv_hiwat_minmss tcps_propinfo_tbl[32].prop_cur_uval

```

```

464 #define tcps_fin_wait_2_flush_interval_high \
465         tcps_propinfo_tbl[33].prop_max_uval
466 #define tcps_fin_wait_2_flush_interval_low \
467         tcps_propinfo_tbl[33].prop_cur_uval
468 #define tcps_fin_wait_2_flush_interval_min \
469         tcps_propinfo_tbl[33].prop_min_uval
470 #define tcps_max_buf \
471         tcps_propinfo_tbl[34].prop_cur_uval
472 #define tcps_strong_iss \
473         tcps_propinfo_tbl[35].prop_cur_uval
474 #define tcps_rtt_updates \
475         tcps_propinfo_tbl[36].prop_cur_uval
476 #define tcps_wscales_always \
477         tcps_propinfo_tbl[37].prop_cur_bval
478 #define tcps_tstamp_always \
479         tcps_propinfo_tbl[38].prop_cur_bval
480 #define tcps_tstamp_if_wscales \
481         tcps_propinfo_tbl[39].prop_cur_bval
482 #define tcps_rexmit_interval_extra \
483         tcps_propinfo_tbl[40].prop_cur_uval
484 #define tcps_deferred_acks_max \
485         tcps_propinfo_tbl[41].prop_cur_uval
486 #define tcps_slow_start_after_idle \
487         tcps_propinfo_tbl[42].prop_cur_uval
488 #define tcps_slow_start_initial \
489         tcps_propinfo_tbl[43].prop_cur_uval
490 #define tcps_sack_permitted \
491         tcps_propinfo_tbl[44].prop_cur_uval
492 #define tcps_ipv6_hoplimit \
493         tcps_propinfo_tbl[45].prop_cur_uval
494 #define tcps_mss_def_ipv6 \
495         tcps_propinfo_tbl[46].prop_cur_uval
496 #define tcps_mss_max_ipv6 \
497         tcps_propinfo_tbl[47].prop_cur_uval
498 #define tcps_rev_src_routes \
499         tcps_propinfo_tbl[48].prop_cur_bval
500 #define tcps_local_dack_interval \
501         tcps_propinfo_tbl[49].prop_cur_uval
502 #define tcps_local_dacks_max \
503         tcps_propinfo_tbl[50].prop_cur_uval
504 #define tcps_ecn_permitted \
505         tcps_propinfo_tbl[51].prop_cur_uval
506 #define tcps_rst_sent_rate_enabled \
507         tcps_propinfo_tbl[52].prop_cur_bval
508 #define tcps_rst_sent_rate \
509         tcps_propinfo_tbl[53].prop_cur_uval
510 #define tcps_push_timer_interval \
511         tcps_propinfo_tbl[54].prop_cur_uval
512 #define tcps_use_smss_as_mss_opt \
513         tcps_propinfo_tbl[55].prop_cur_bval
514 #define tcps_keepalive_abort_interval_high \
515         tcps_propinfo_tbl[56].prop_max_uval
516 #define tcps_keepalive_abort_interval_low \
517         tcps_propinfo_tbl[56].prop_min_uval
518 #define tcps_keepalive_abort_interval \
519         tcps_propinfo_tbl[56].prop_cur_uval
520 #define tcps_wroff_xtra \
521         tcps_propinfo_tbl[57].prop_cur_uval
522 #define tcps_dev_flow_ctl \
523         tcps_propinfo_tbl[58].prop_cur_bval
524 #define tcps_reass_timeout \
525         tcps_propinfo_tbl[59].prop_cur_uval
526 #define tcps_iss_incr \
527         tcps_propinfo_tbl[65].prop_cur_uval
528
529 extern struct qinit tcp_rinitv4, tcp_rinitv6;
530 extern boolean_t do_tcp_fusion;
531
532 /*
533  * Object to represent database of options to search passed to
534  * {sock,tpi}optcom_req() interface routine to take care of option
535  * management and associated methods.
536  */
537 extern optdb_obj_t      tcp_opt_obj;
538 extern uint_t          tcp_max_optsize;
539
540 extern int tcp_queue_flag;
541
542 extern uint_t tcp_free_list_max_cnt;
543
544 /*
545  * Functions in tcp.c.
546  */
547 extern void tcp_acceptor_hash_insert(t_uscalar_t, tcp_t *);
548 extern tcp_t *tcp_acceptor_hash_lookup(t_uscalar_t, tcp_stack_t *);
549 extern void tcp_acceptor_hash_remove(tcp_t *);
550 extern mblk_t *tcp_ack_mp(tcp_t *);
551 extern int tcp_build_hdrs(tcp_t *);
552 extern void tcp_cleanup(tcp_t *);
553 extern int tcp_clean_death(tcp_t *, int);
554 extern void tcp_clean_death_wrapper(void *, mblk_t *, void *,
555         ip_rcv_attr_t *);
556 extern void tcp_close_common(conn_t *, int);

```

```

530 extern void tcp_close_detached(tcp_t *);
531 extern void tcp_close_mpp(mblk_t **);
532 extern void tcp_closei_local(tcp_t *);
533 extern sock_lower_handle_t tcp_create(int, int, int, sock_downcalls_t **,
534         uint_t *, int *, int, cred_t *);
535 extern conn_t *tcp_create_common(cred_t *, boolean_t, boolean_t, int *);
536 extern void tcp_disconnect(tcp_t *, mblk_t *);
537 extern char *tcp_display(tcp_t *, char *, char);
538 extern int tcp_do_bind(conn_t *, struct sockaddr *, socklen_t, cred_t *,
539         boolean_t);
540 extern int tcp_do_connect(conn_t *, const struct sockaddr *, socklen_t,
541         cred_t *, pid_t);
542 extern int tcp_do_listen(conn_t *, struct sockaddr *, socklen_t, int,
543         cred_t *, boolean_t);
544 extern int tcp_do_unbind(conn_t *);
545 extern boolean_t tcp_eager_blowoff(tcp_t *, t_scalar_t);
546 extern void tcp_eager_cleanup(tcp_t *, boolean_t);
547 extern void tcp_eager_kill(void *, mblk_t *, void *, ip_rcv_attr_t *);
548 extern void tcp_eager_unlink(tcp_t *);
549 extern int tcp_getpeername(sock_lower_handle_t, struct sockaddr *,
550         socklen_t *, cred_t *);
551 extern int tcp_getsockname(sock_lower_handle_t, struct sockaddr *,
552         socklen_t *, cred_t *);
553 extern void tcp_init_values(tcp_t *, tcp_t *);
554 extern void tcp_ipsec_cleanup(tcp_t *);
555 extern int tcp_maxpsz_set(tcp_t *, boolean_t);
556 extern void tcp_mss_set(tcp_t *, uint32_t);
557 extern void tcp_reinput(conn_t *, mblk_t *, ip_rcv_attr_t *, ip_stack_t *);
558 extern void tcp_rsrv(queue_t *);
559 extern uint_t tcp_rwnd_reopen(tcp_t *);
560 extern int tcp_rwnd_set(tcp_t *, uint32_t);
561 extern int tcp_set_destination(tcp_t *);
562 extern void tcp_set_ws_value(tcp_t *);
563 extern void tcp_stop_lingering(tcp_t *);
564 extern void tcp_update_pmtu(tcp_t *, boolean_t);
565 extern mblk_t *tcp_zcopy_backoff(tcp_t *, mblk_t *, boolean_t);
566 extern boolean_t tcp_zcopy_check(tcp_t *);
567 extern void tcp_zcopy_notify(tcp_t *);
568 extern void tcp_get_proto_props(tcp_t *, struct sock_proto_props *);
569
570 /*
571  * Bind related functions in tcp_bind.c
572  */
573 extern int tcp_bind_check(conn_t *, struct sockaddr *, socklen_t,
574         cred_t *, boolean_t);
575 extern void tcp_bind_hash_insert(tf_t *, tcp_t *, int);
576 extern void tcp_bind_hash_remove(tcp_t *);
577 extern in_port_t tcp_bindi(tcp_t *, in_port_t, const in6_addr_t *,
578         int, boolean_t, boolean_t, boolean_t);
579 extern in_port_t tcp_update_next_port(in_port_t, const tcp_t *,
580         boolean_t);
581
582 /*
583  * Fusion related functions in tcp_fusion.c.
584  */
585 extern void tcp_fuse(tcp_t *, uchar_t *, tpha_t *);
586 extern void tcp_unfuse(tcp_t *);
587 extern boolean_t tcp_fuse_output(tcp_t *, mblk_t *, uint32_t);
588 extern void tcp_fuse_output_urg(tcp_t *, mblk_t *);
589 extern boolean_t tcp_fuse_rcv_drain(queue_t *, tcp_t *, mblk_t **);
590 extern size_t tcp_fuse_set_rcv_hiwat(tcp_t *, size_t);
591 extern int tcp_fuse_maxpsz(tcp_t *);
592 extern void tcp_fuse_backenable(tcp_t *);
593 extern void tcp_iss_key_init(uint8_t *, int, tcp_stack_t *);
594
595 /*

```

```

592 * Output related functions in tcp_output.c.
593 */
594 extern void tcp_close_output(void *, mblk_t *, void *, ip_rcv_attr_t *);
595 extern void tcp_output(void *, mblk_t *, void *, ip_rcv_attr_t *);
596 extern void tcp_output_urgent(void *, mblk_t *, void *, ip_rcv_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_rcv_attr_t *);
601 extern void tcp_shutdown_output(void *, mblk_t *, void *, ip_rcv_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_rcv_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_rcv_attr_t *);
618 extern void tcp_input_data(void *, mblk_t *, void *, ip_rcv_attr_t *);
619 extern void tcp_input_listener_unbound(void *, mblk_t *, void *,
620 ip_rcv_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 *, icmp6_t *, icmp6_t *,
625 ip_rcv_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_rcv_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_keeplive_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 *), hrttime_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_rcv_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_rcv_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_rcv_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

new/usr/src/uts/common/inet/tcp_stats.h

1

```
*****
7689 Wed Jun 13 12:05:30 2012
new/usr/src/uts/common/inet/tcp_stats.h
%B
*****
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) 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 #ifndef _INET_TCP_STATS_H
27 #define _INET_TCP_STATS_H

29 /*
30  * TCP private kernel statistics declarations.
31 */

33 #ifdef __cplusplus
34 extern "C" {
35 #endif

37 #ifdef _KERNEL

39 /*
40  * TCP Statistics.
41  *
42  * How TCP statistics work.
43  *
44  * There are two types of statistics invoked by two macros.
45  *
46  * TCP_STAT(name) does non-atomic increment of a named stat counter. It is
47  * supposed to be used in non MT-hot paths of the code.
48  *
49  * TCP_DBGSTAT(name) does atomic increment of a named stat counter. It is
50  * supposed to be used for DEBUG purposes and may be used on a hot path.
51  * These counters are only available in a debugged kernel. They are grouped
52  * These counters are only available in a debugged kernel. They are grouped
53  * under the TCP_DEBUG_COUNTER C pre-processor condition.
54  *
55  * Both TCP_STAT and TCP_DBGSTAT counters are available using kstat
56  * (use "kstat tcp" to get them).
57  *
58  * How to add new counters.
59  *
60  * 1) Add a field in the tcp_stat structure describing your counter.
61  * 2) Add a line in the template in tcp_kstat2_init() with the name
```

new/usr/src/uts/common/inet/tcp_stats.h

2

```
61  * of the counter.
62  * 3) Update tcp_clr_stats() and tcp_cp_stats() with the new counters.
63  * IMPORTANT!! - make sure that all the above functions are in sync !!
64  * 4) Use either TCP_STAT or TCP_DBGSTAT with the name.
65  *
66  * Please avoid using private counters which are not kstat-exported.
67  *
68  * Implementation note.
69  *
70  * Both the MIB2 and tcp_stat_t counters are kept per CPU in the array
71  * tcps_sc in tcp_stack_t. Each array element is a pointer to a
72  * tcp_stats_cpu_t struct. Once allocated, the tcp_stats_cpu_t struct is
73  * not freed until the tcp_stack_t is going away. So there is no need to
74  * acquire a lock before accessing the stats counters.
75 */

77 #ifndef TCP_DEBUG_COUNTER
78 #ifdef DEBUG
79 #define TCP_DEBUG_COUNTER 1
80 #else
81 #define TCP_DEBUG_COUNTER 0
82 #endif
83 #endif

85 /* Kstats */
86 typedef struct tcp_stat {
87     kstat_named_t    tcp_time_wait_syn_success;
88     kstat_named_t    tcp_clean_death_nondetached;
89     kstat_named_t    tcp_eager_blowoff_q;
90     kstat_named_t    tcp_eager_blowoff_q0;
91     kstat_named_t    tcp_no_listener;
92     kstat_named_t    tcp_listendrop;
93     kstat_named_t    tcp_listendropq0;
94     kstat_named_t    tcp_wsrv_called;
95     kstat_named_t    tcp_flwctl_on;
96     kstat_named_t    tcp_timer_fire_early;
97     kstat_named_t    tcp_timer_fire_miss;
98     kstat_named_t    tcp_zcopy_on;
99     kstat_named_t    tcp_zcopy_off;
100    kstat_named_t    tcp_zcopy_backoff;
101    kstat_named_t    tcp_fusion_flowctl;
102    kstat_named_t    tcp_fusion_backenabed;
103    kstat_named_t    tcp_fusion_urg;
104    kstat_named_t    tcp_fusion_putnext;
105    kstat_named_t    tcp_fusion_unfusable;
106    kstat_named_t    tcp_fusion_aborted;
107    kstat_named_t    tcp_fusion_unqualified;
108    kstat_named_t    tcp_fusion_rrw_busy;
109    kstat_named_t    tcp_fusion_rrw_msgcnt;
110    kstat_named_t    tcp_fusion_rrw_plugged;
111    kstat_named_t    tcp_in_ack_unsent_drop;
112    kstat_named_t    tcp_sock_fallback;
113    kstat_named_t    tcp_lso_enabled;
114    kstat_named_t    tcp_lso_disabled;
115    kstat_named_t    tcp_lso_times;
116    kstat_named_t    tcp_lso_pkt_out;
117    kstat_named_t    tcp_listen_cnt_drop;
118    kstat_named_t    tcp_listen_mem_drop;
119    kstat_named_t    tcp_zwin_mem_drop;
120    kstat_named_t    tcp_zwin_ack_syn;
121    kstat_named_t    tcp_rst_unsent;
122    kstat_named_t    tcp_reclaim_cnt;
123    kstat_named_t    tcp_reass_timeout;
124 #ifdef TCP_DEBUG_COUNTER
125     kstat_named_t    tcp_time_wait;
126     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 Wed Jun 13 12:05:32 2012
new/usr/src/uts/common/inet/tunables.c
%B
*****
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 */
21 /*
22 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 1990 Mentat Inc.
24 */

26 #include <inet/tunables.h>
27 #include <sys/md5.h>
28 #include <inet/common.h>
29 #include <inet/ip.h>
30 #include <inet/ip6.h>
31 #include <netinet/icmp6.h>
32 #include <inet/ip_stack.h>
33 #include <inet/rawip_impl.h>
34 #include <inet/tcp_stack.h>
35 #include <inet/tcp_impl.h>
36 #include <inet/udp_impl.h>
37 #include <inet/dccp/dccp_stack.h>
38 #include <inet/dccp/dccp_impl.h>
39 #endif /* ! codereview */
40 #include <inet/sctp/sctp_stack.h>
41 #include <inet/sctp/sctp_impl.h>
42 #include <inet/tunables.h>

44 static int
45 prop_perm2const(mod_prop_info_t *pinfo)
46 {
47     if (pinfo->mpi_setf == NULL)
48         return (MOD_PROP_PERM_READ);
49     if (pinfo->mpi_getf == NULL)
50         return (MOD_PROP_PERM_WRITE);
51     return (MOD_PROP_PERM_RW);
52 }

54 /*
55  * Modifies the value of the property to default value or to the 'pval'
56  * specified by the user.
57  */
58 /* ARGSUSED */
59 int
60 mod_set_boolean(void *cbarg, cred_t *cr, mod_prop_info_t *pinfo,
61                const char *ifname, const void* pval, uint_t flags)

```

```

62 {
63     char *end;
64     unsigned long new_value;

66     if (flags & MOD_PROP_DEFAULT) {
67         pinfo->prop_cur_bval = pinfo->prop_def_bval;
68         return (0);
69     }

71     if (ddi_strtoul(pval, &end, 10, &new_value) != 0 || *end != '\0')
72         return (EINVAL);
73     if (new_value != B_TRUE && new_value != B_FALSE)
74         return (EINVAL);
75     pinfo->prop_cur_bval = new_value;
76     return (0);
77 }

79 /*
80  * Retrieves property permission, default value, current value or possible
81  * values for those properties whose value type is boolean_t.
82  */
83 /* ARGSUSED */
84 int
85 mod_get_boolean(void *cbarg, mod_prop_info_t *pinfo, const char *ifname,
86                void *pval, uint_t psize, uint_t flags)
87 {
88     boolean_t get_def = (flags & MOD_PROP_DEFAULT);
89     boolean_t get_perm = (flags & MOD_PROP_PERM);
90     boolean_t get_range = (flags & MOD_PROP_POSSIBLE);
91     size_t nbytes;

93     bzero(pval, psize);
94     if (get_perm)
95         nbytes = snprintf(pval, psize, "%u", prop_perm2const(pinfo));
96     else if (get_range)
97         nbytes = snprintf(pval, psize, "%u,%u", B_FALSE, B_TRUE);
98     else if (get_def)
99         nbytes = snprintf(pval, psize, "%u", pinfo->prop_def_bval);
100    else
101        nbytes = snprintf(pval, psize, "%u", pinfo->prop_cur_bval);
102    if (nbytes >= psize)
103        return (ENOBUFS);
104    return (0);
105 }

107 int
108 mod_uint32_value(const void *pval, mod_prop_info_t *pinfo, uint_t flags,
109                 ulong_t *new_value)
110 {
111     char *end;

113     if (flags & MOD_PROP_DEFAULT) {
114         *new_value = pinfo->prop_def_uval;
115         return (0);
116     }

118     if (ddi_strtoul(pval, &end, 10, (ulong_t *)new_value) != 0 ||
119         *end != '\0')
120         return (EINVAL);
121     if (*new_value < pinfo->prop_min_uval ||
122         *new_value > pinfo->prop_max_uval) {
123         return (ERANGE);
124     }
125     return (0);
126 }

```

```

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

```

```

392 */
393 /* ARGSUSED */
394 int
395 mod_get_extra_privports(void *cbarg, mod_prop_info_t *pinfo, const char *ifname,
396 void *val, uint_t psize, uint_t flags)
397 {
398     uint_t      proto = pinfo->mpi_proto;
399     tcp_stack_t *tcps;
400     sctp_stack_t *sctps;
401     udp_stack_t *us;
402     uint_t      i, nports, size;
403     in_port_t   *ports;
404     char        *pval = val;
405     size_t      nbytes = 0, tbytes = 0;
406     boolean_t   get_def = (flags & MOD_PROP_DEFAULT);
407     boolean_t   get_perm = (flags & MOD_PROP_PERM);
408     boolean_t   get_range = (flags & MOD_PROP_POSSIBLE);
409
410     bzero(pval, psize);
411     size = psize;
412
413     if (get_def) {
414         tbytes = snprintf(pval, psize, "%u,%u", ULP_DEF_EPRIV_PORT1,
415             ULP_DEF_EPRIV_PORT2);
416         goto ret;
417     } else if (get_perm) {
418         tbytes = snprintf(pval, psize, "%u", MOD_PROP_PERM_RW);
419         goto ret;
420     }
421
422     switch (proto) {
423     case MOD_PROTO_TCP:
424         tcps = (tcp_stack_t *)cbarg;
425         ports = tcps->tcps_g_epriv_ports;
426         nports = tcps->tcps_g_num_epriv_ports;
427         break;
428     case MOD_PROTO_UDP:
429         us = (udp_stack_t *)cbarg;
430         ports = us->us_epriv_ports;
431         nports = us->us_num_epriv_ports;
432         break;
433     case MOD_PROTO_SCTP:
434         sctps = (sctp_stack_t *)cbarg;
435         ports = sctps->sctps_g_epriv_ports;
436         nports = sctps->sctps_g_num_epriv_ports;
437         break;
438     default:
439         return (ENOTSUP);
440     }
441
442     if (get_range) {
443         tbytes = snprintf(pval, psize, "%u-%u", pinfo->prop_min_uval,
444             pinfo->prop_max_uval);
445         goto ret;
446     }
447
448     for (i = 0; i < nports; i++) {
449         if (ports[i] != 0) {
450             if (psize == size)
451                 nbytes = snprintf(pval, size, "%u", ports[i]);
452             else
453                 nbytes = snprintf(pval, size, ",%u", ports[i]);
454             size -= nbytes;
455             pval += nbytes;
456             tbytes += nbytes;
457             if (tbytes >= psize)

```

```

458         return (ENOBUFS);
459     }
460     }
461     return (0);
462 ret:
463     if (tbytes >= psize)
464         return (ENOBUFS);
465     return (0);
466 }

```

```

*****
6181 Wed Jun 13 12:05:33 2012
new/usr/src/uts/common/inet/tunables.h
%B
*****
_____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_uamin;
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_uamin
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 */
```


new/usr/src/uts/common/netinet/dccp.h

1

795 Wed Jun 13 12:05:35 2012

new/usr/src/uts/common/netinet/dccp.h

%B

```
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2012 David Hoepfner. All rights reserved.
14 */

16 #ifndef _NETINET_DCCP_H
17 #define _NETINET_DCCP_H

19 #ifdef __cplusplus
20 extern "C" {
21 #endif

23 /*
24  * DCCP states
25 */
26 #define DCCPS_CLOSED -5
27 #define DCCPS_BOUND -4
28 #define DCCPS_LISTEN -3
29 #define DCCPS_REQUEST -2
30 #define DCCPS_RESPOND -1
31 #define DCCPS_PARTOPEN 0

34 #ifdef __cplusplus
35 }
36 #endif

38 #endif /* _NETINET_DCCP_H */
39 #endif /* !codereview */
```

```

*****
44002 Wed Jun 13 12:05:36 2012
new/usr/src/uts/common/netinet/in.h
%B
*****
_____unchanged_portion_omitted_____
128 #define s6_addr      _S6_un._S6_u8

130 #ifndef _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 #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_PRINTSERV 515
244 #define IPPORT_EFSERVER  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
255
256 /*
257  * DHCPv6 UDP ports
258  */
259 #define IPPORT_DHCPV6C        546
260 #define IPPORT_DHCPV6S        547
261
262 #define IPPORT_SOCKETS        1080
263
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
272
273 /*
274  * Link numbers
275  */
276 #define IMPLINK_IP            155
277 #define IMPLINK_LOWEXPER      156
278 #define IMPLINK_HIGHEXPER     158
279
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  */
287
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__) */
294
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         uint32_t _S_addr;
300     };
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 };
312
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        0xffff0000U
325 #define IN_CLASSA_NSHIFT     24
326 #define IN_CLASSA_HOST      0x00ffffffU
327 #define IN_CLASSA_MAX        128
328
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
334
335 #define IN_CLASSC(i)          (((i) & 0xe0000000U) == 0xc0000000U)
336 #define IN_CLASSC_NET        0xfffff000U
337 #define IN_CLASSC_NSHIFT     8
338 #define IN_CLASSC_HOST      0x000000ffU
339
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      0x0ffffffU /* routing needn't know */
344
345 #define IN_CLASSE(i)          (((i) & 0xf0000000U) == 0xf0000000U)
346 #define IN_CLASSE_NET        0xffffffffU
347
348 #define IN_MULTICAST(i)      IN_CLASSD(i)
349
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
359
360 #define INADDR_ANY            0x00000000U
361 #define INADDR_LOOPBACK      0x7f000001U
362 #define INADDR_BROADCAST     0xffffffffU /* must be masked */
363 #define INADDR_NONE          0xffffffffU
364
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 */
370
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    0xffff0000U
376 #define IN_PRIVATE12_NET    0xac100000U /* 172.16/12 */
377 #define IN_PRIVATE12_MASK   0xffff0000U
378 #define IN_PRIVATE16_NET    0xc0a80000U /* 192.168/16 */
379 #define IN_PRIVATE16_MASK   0xffff0000U
380
381 /* RFC 3927 IPv4 link local address (i in host byte-order) */
382 #define IN_LINKLOCAL(i)     (((i) & IN_AUTOCONF_MASK) == IN_AUTOCONF_NET)
383
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 #ifndef _BIG_ENDIAN

434 /* masks */
435 #define IPV6_FLOWINFO_FLOWLABEL 0x000fffffu
436 #define IPV6_FLOWINFO_TCLASS 0x0fff0000u

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 #define IN6_IS_ADDR_UNSPECIFIED (const struct in6_addr *);
495 #define IN6_IS_ADDR_LOOPBACK (const struct in6_addr *);
496 #define IN6_IS_ADDR_MULTICAST (const struct in6_addr *);
497 #define IN6_IS_ADDR_LINKLOCAL (const struct in6_addr *);
498 #define IN6_IS_ADDR_SITELOCAL (const struct in6_addr *);
499 #define IN6_IS_ADDR_V4MAPPED (const struct in6_addr *);
500 #define IN6_IS_ADDR_V4MAPPED_ANY (const struct in6_addr *); -- Not from RFC2553
501 #define IN6_IS_ADDR_MC_GLOBAL (const struct in6_addr *);
502 #define IN6_IS_ADDR_MC_RESERVED (const struct in6_addr *); -- Not from RFC2553
503 #define IN6_IS_ADDR_MC_NODELOCAL (const struct in6_addr *);
504 #define IN6_IS_ADDR_MC_LINKLOCAL (const struct in6_addr *);
505 #define IN6_IS_ADDR_MC_SITELOCAL (const struct in6_addr *);
506 #define IN6_IS_ADDR_MC_ORGLOCAL (const struct in6_addr *);
507 #define IN6_IS_ADDR_MC_GLOBAL (const struct in6_addr *);
508 #define IN6_IS_ADDR_6TO4 (const struct in6_addr *); -- Not from RFC2553
509 #define IN6_IS_ADDR_6TO4_PREFIX_EQUAL (const struct in6_addr *,
510 const struct in6_addr *); -- Not from RFC2553
511 #define IN6_IS_ADDR_LINKSCOPE (const struct in6_addr *); -- 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 #ifndef _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 #ifndef _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 #ifndef _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] & 0x000c0fff) == 0x000080fe)
548 #endif /* _BIG_ENDIAN */

550 #ifndef _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] & 0x000c0fff) == 0x0000c0fe)
556 #endif /* _BIG_ENDIAN */

558 #ifndef _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] == 0xffff0000) && \
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 #ifndef _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] == 0xffff0000) && \
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 #ifndef _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 #ifndef _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 #ifndef _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 #ifndef _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 #ifndef _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 #ifndef _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 ff02:0:0:0:1: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 */
717 */
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 bitwise copy; the V4ADDR in the 6to4 address is not
781  * 32-bit aligned.

```

```

782 *
783 * void IN6_6TO4_TO_V4ADDR(const in6_addr_t *v6, struct in_addr *v4);
784 *
785 */
786 #define IN6_6TO4_TO_V4ADDR(v6, v4) \
787 ((v4)->_S_un._S_un_b.s_b1 = (v6)->_S6_un._S6_u8[2], \
788 (v4)->_S_un._S_un_b.s_b2 = (v6)->_S6_un._S6_u8[3], \
789 (v4)->_S_un._S_un_b.s_b3 = (v6)->_S6_un._S6_u8[4], \
790 (v4)->_S_un._S_un_b.s_b4 = (v6)->_S6_un._S6_u8[5])
791
792 /*
793 * IN6_V4ADDR_TO_6TO4
794 * Given an IPv4 address and an IPv6 address for output, a 6to4 address
795 * will be created from the IPv4 Address.
796 * Note: This method for creating 6to4 addresses is not standardized
797 * outside of Solaris. The newly created 6to4 address will be of the form
798 * 2002:<V4ADDR>:<SUBNETID>:<HOSTID>, where SUBNETID will equal 0 and
799 * HOSTID will equal 1.
800 *
801 * void IN6_V4ADDR_TO_6TO4(const struct in_addr *v4, in6_addr_t *v6)
802 *
803 */
804 #ifndef _BIG_ENDIAN
805 #define IN6_V4ADDR_TO_6TO4(v4, v6) \
806 ((v6)->_S6_un._S6_u8[0] = 0x20, \
807 (v6)->_S6_un._S6_u8[1] = 0x02, \
808 (v6)->_S6_un._S6_u8[2] = (v4)->_S_un._S_un_b.s_b1, \
809 (v6)->_S6_un._S6_u8[3] = (v4)->_S_un._S_un_b.s_b2, \
810 (v6)->_S6_un._S6_u8[4] = (v4)->_S_un._S_un_b.s_b3, \
811 (v6)->_S6_un._S6_u8[5] = (v4)->_S_un._S_un_b.s_b4, \
812 (v6)->_S6_un._S6_u8[6] = 0, \
813 (v6)->_S6_un._S6_u8[7] = 0, \
814 (v6)->_S6_un._S6_u32[2] = 0, \
815 (v6)->_S6_un._S6_u32[3] = 0x00000001U)
816 #else
817 #define IN6_V4ADDR_TO_6TO4(v4, v6) \
818 ((v6)->_S6_un._S6_u8[0] = 0x20, \
819 (v6)->_S6_un._S6_u8[1] = 0x02, \
820 (v6)->_S6_un._S6_u8[2] = (v4)->_S_un._S_un_b.s_b1, \
821 (v6)->_S6_un._S6_u8[3] = (v4)->_S_un._S_un_b.s_b2, \
822 (v6)->_S6_un._S6_u8[4] = (v4)->_S_un._S_un_b.s_b3, \
823 (v6)->_S6_un._S6_u8[5] = (v4)->_S_un._S_un_b.s_b4, \
824 (v6)->_S6_un._S6_u8[6] = 0, \
825 (v6)->_S6_un._S6_u8[7] = 0, \
826 (v6)->_S6_un._S6_u32[2] = 0, \
827 (v6)->_S6_un._S6_u32[3] = 0x01000000U)
828 #endif /* _BIG_ENDIAN */
829
830 /*
831 * IN6_ARE_ADDR_EQUAL (defined in RFC2292)
832 * Compares if IPv6 addresses are equal.
833 * Note: Compares in order of high likelihood of a miss so we minimize
834 * compares. (Current heuristic order, compare in reverse order of
835 * uint32_t units)
836 *
837 * bool IN6_ARE_ADDR_EQUAL(const struct in6_addr *,
838 *                          const struct in6_addr *);
839 */
840 #define IN6_ARE_ADDR_EQUAL(addr1, addr2) \
841 (((addr1)->_S6_un._S6_u32[3] == (addr2)->_S6_un._S6_u32[3]) && \
842 ((addr1)->_S6_un._S6_u32[2] == (addr2)->_S6_un._S6_u32[2]) && \
843 ((addr1)->_S6_un._S6_u32[1] == (addr2)->_S6_un._S6_u32[1]) && \
844 ((addr1)->_S6_un._S6_u32[0] == (addr2)->_S6_un._S6_u32[0]))
845
846 /*
847 * IN6_ARE_PREFIXEDADDR_EQUAL (not defined in RFCs)

```

```

848 * Compares if prefixed parts of IPv6 addresses are equal.
849 *
850 * uint32_t IN6_MASK_FROM_PREFIX(int, int);
851 * bool IN6_ARE_PREFIXEDADDR_EQUAL(const struct in6_addr *,
852 *                                 const struct in6_addr *,
853 *                                 int);
854 */
855 #define IN6_MASK_FROM_PREFIX(qoctet, prefix) \
856 (((qoctet) + 1) * 32 < (prefix)) ? 0xFFFFFFFFu : \
857 (((qoctet) * 32) >= (prefix)) ? 0x00000000u : \
858 0xFFFFFFFFu << (((qoctet) + 1) * 32 - (prefix)))
859
860 #define IN6_ARE_PREFIXEDADDR_EQUAL(addr1, addr2, prefix) \
861 (((ntohl((addr1)->_S6_un._S6_u32[0]) & \
862 IN6_MASK_FROM_PREFIX(0, prefix)) == \
863 (ntohl((addr2)->_S6_un._S6_u32[0]) & \
864 IN6_MASK_FROM_PREFIX(0, prefix))) && \
865 ((ntohl((addr1)->_S6_un._S6_u32[1]) & \
866 IN6_MASK_FROM_PREFIX(1, prefix)) == \
867 (ntohl((addr2)->_S6_un._S6_u32[1]) & \
868 IN6_MASK_FROM_PREFIX(1, prefix))) && \
869 ((ntohl((addr1)->_S6_un._S6_u32[2]) & \
870 IN6_MASK_FROM_PREFIX(2, prefix)) == \
871 (ntohl((addr2)->_S6_un._S6_u32[2]) & \
872 IN6_MASK_FROM_PREFIX(2, prefix))) && \
873 ((ntohl((addr1)->_S6_un._S6_u32[3]) & \
874 IN6_MASK_FROM_PREFIX(3, prefix)) == \
875 (ntohl((addr2)->_S6_un._S6_u32[3]) & \
876 IN6_MASK_FROM_PREFIX(3, prefix))))
877
878 #endif /* !defined(_XPG4_2) || defined(_XPG6) || defined(__EXTENSIONS__) */
879
880 /*
881 * Options for use with [gs]etsockopt at the IP level.
882 * Note: Some of the IP_namespace has conflict with and
883 * and is exposed through <xti.h>. (It also requires exposing
884 * options not implemented. The options with potential
885 * for conflicts use #ifndef guards.
886 */
887 #ifndef IP_OPTIONS
888 #define IP_OPTIONS 1 /* set/get IP per-packet options */
889 #endif
890
891 #define IP_HDRINCL 2 /* int; header is included with data (raw) */
892
893 #define IP_TOS 3 /* int; IP type of service and precedence */
894 #endif
895
896 #ifndef IP_TTL
897 #define IP_TTL 4 /* int; IP time to live */
898 #endif
899
900 #define IP_RECVOPTS 0x5 /* int; receive all IP options w/datagram */
901 #define IP_RECVTOPTS 0x6 /* int; receive IP options for response */
902 #define IP_RECVSTADDR 0x7 /* int; receive IP dst addr w/datagram */
903 #define IP_RETOPTS 0x8 /* ip_opts; set/get IP per-packet options */
904 #define IP_RECVIF 0x9 /* int; receive the inbound interface index */
905 #define IP_RECVSLLA 0xa /* sockaddr_dl; get source link layer address */
906 #define IP_RECVTTL 0xb /* uint8_t; get TTL for inbound packet */
907
908 #define IP_MULTICAST_IF 0x10 /* set/get IP multicast interface */
909 #define IP_MULTICAST_TTL 0x11 /* set/get IP multicast timetolive */
910 #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 */
973 #define IP_DHCPINIT_IF      0x44 /* accept all unicast DHCP traffic */
974 #define IP_DHCPINIT_IF     0x45

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 #define T_IP_OPTIONS        0x107 /* IP per-packet options */
994 #define 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/rcv 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/rcv 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_pktinfo" - */
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;

```

```
1310 #else
1311 extern struct in6_addr in6addr_any;
1312 extern struct in6_addr in6addr_loopback;
1313 #endif
1314 #endif
1315 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

1317 #ifdef __cplusplus
1318 }
1319 #endif

1321 #endif /* _NETINET_IN_H */
```

new/usr/src/uts/common/sys/netstack.h

1

```
*****
9019 Wed Jun 13 12:05:41 2012
new/usr/src/uts/common/sys/netstack.h
%B
*****
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 2009 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */
26 #ifndef _SYS_NETSTACK_H
27 #define _SYS_NETSTACK_H
29 #include <sys/kstat.h>
31 #ifdef __cplusplus
32 extern "C" {
33 #endif
35 /*
36  * This allows various pieces in and around IP to have a separate instance
37  * for each instance of IP. This is used to support zones that have an
38  * exclusive stack.
39  * Pieces of software far removed from IP (e.g., kernel software
40  * sitting on top of TCP or UDP) probably should not use the netstack
41  * support; if such software wants to support separate zones it
42  * can do that using the zones framework (zone_key_create() etc)
43  * whether there is a shared IP stack or an exclusive IP stack underneath.
44  */
46 /*
47  * Each netstack has an identifier. We reuse the zoneid allocation for
48  * this but have a separate typedef. Thus the shared stack (used by
49  * the global zone and other shared stack zones) have a zero ID, and
50  * the exclusive stacks have a netstackid that is the same as their zoneid.
51  */
52 typedef id_t netstackid_t;
54 #define GLOBAL_NETSTACKID 0
56 /*
57  * One for each module which uses netstack support.
58  * Used in netstack_register().
59  *
60  * The order of these is important for some modules both for
61  * the creation (which done in ascending order) and destruction (which is
```

new/usr/src/uts/common/sys/netstack.h

2

```
62 * done in in decending order).
63 */
64 #define NS_ALL -1 /* Match all */
65 #define NS_DLS 0
66 #define NS_IPTUN 1
67 #define NS_STR 2 /* autopush list etc */
68 #define NS_HOOK 3
69 #define NS_NETI 4
70 #define NS_ARP 5
71 #define NS_IP 6
72 #define NS_ICMP 7
73 #define NS_UDP 8
74 #define NS_TCP 9
75 #define NS_SCTP 10
76 #define NS_RTS 11
77 #define NS_IPSEC 12
78 #define NS_KEYSOCK 13
79 #define NS_SPDsock 14
80 #define NS_IPSECAH 15
81 #define NS_IPSECESP 16
82 #define NS_IPNET 17
83 #define NS_ILB 18
84 #define NS_DCCP 19
85 #define NS_MAX (NS_DCCP+1)
84 #define NS_MAX (NS_ILB+1)
87 /*
88  * State maintained for each module which tracks the state of
89  * the create, shutdown and destroy callbacks.
90  *
91  * Keeps track of pending actions to avoid holding locks when
92  * calling into the create/shutdown/destroy functions in the module.
93  */
94 #ifdef _KERNEL
95 typedef struct {
96     uint16_t nms_flags;
97     kcondvar_t nms_cv;
98 } nm_state_t;
100 /*
101  * nms_flags
102  */
103 #define NSS_CREATE_NEEDED 0x0001
104 #define NSS_CREATE_INPROGRESS 0x0002
105 #define NSS_CREATE_COMPLETED 0x0004
106 #define NSS_SHUTDOWN_NEEDED 0x0010
107 #define NSS_SHUTDOWN_INPROGRESS 0x0020
108 #define NSS_SHUTDOWN_COMPLETED 0x0040
109 #define NSS_DESTROY_NEEDED 0x0100
110 #define NSS_DESTROY_INPROGRESS 0x0200
111 #define NSS_DESTROY_COMPLETED 0x0400
113 #define NSS_CREATE_ALL \
114     (NSS_CREATE_NEEDED|NSS_CREATE_INPROGRESS|NSS_CREATE_COMPLETED)
115 #define NSS_SHUTDOWN_ALL \
116     (NSS_SHUTDOWN_NEEDED|NSS_SHUTDOWN_INPROGRESS|NSS_SHUTDOWN_COMPLETED)
117 #define NSS_DESTROY_ALL \
118     (NSS_DESTROY_NEEDED|NSS_DESTROY_INPROGRESS|NSS_DESTROY_COMPLETED)
120 #define NSS_ALL_INPROGRESS \
121     (NSS_CREATE_INPROGRESS|NSS_SHUTDOWN_INPROGRESS|NSS_DESTROY_INPROGRESS)
122 #else
123 /* User-level compile like IP Filter needs a netstack_t. Dummy */
124 typedef uint_t nm_state_t;
125 #endif /* _KERNEL */
```

```

127 /*
128 * One for every netstack in the system.
129 * We use a union so that the compiler 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 */
```

new/usr/src/uts/intel/Makefile.intel.shared

1

```
*****
16821 Wed Jun 13 12:05:42 2012
new/usr/src/uts/intel/Makefile.intel.shared
%B
*****
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 # Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
26 #
27 # This makefile contains the common definitions for all intel
28 # implementation architecture independent modules.
29 #
31 #
32 # Machine type (implementation architecture):
33 #
34 PLATFORM = i86pc
36 #
37 # Everybody needs to know how to build modstubs.o and to locate unix.o.
38 # Note that unix.o must currently be selected from among the possible
39 # "implementation architectures". Note further, that unix.o is only
40 # used as an optional error check for undefines so (theoretically)
41 # any "implementation architectures" could be used. We choose i86pc
42 # because it is the reference port.
43 #
44 UNIX_DIR = $(UTSBASE)/i86pc/unix
45 GENLIB_DIR = $(UTSBASE)/intel/genunix
46 IPDRV_DIR = $(UTSBASE)/intel/ip
47 MODSTUBS_DIR = $(UNIX_DIR)
48 DSF_DIR = $(UTSBASE)/$(PLATFORM)/genassym
49 LINTS_DIR = $(OBJS_DIR)
50 LINT_LIB_DIR = $(UTSBASE)/intel/lint-libs/$(OBJS_DIR)
52 UNIX_O = $(UNIX_DIR)/$(OBJS_DIR)/unix.o
53 GENLIB = $(GENLIB_DIR)/$(OBJS_DIR)/libgenunix.so
54 MODSTUBS_O = $(MODSTUBS_DIR)/$(OBJS_DIR)/modstubs.o
55 LINT_LIB = $(UTSBASE)/i86pc/lint-libs/$(OBJS_DIR)/llib-lunix.ln
56 GEN_LINT_LIB = $(UTSBASE)/intel/lint-libs/$(OBJS_DIR)/llib-lgenunix.ln
58 #
59 # Include the makefiles which define build rule templates, the
60 # collection of files per module, and a few specific flags. Note
61 # that order is significant, just as with an include path. The
```

new/usr/src/uts/intel/Makefile.intel.shared

2

```
62 # first build rule template which matches the files name will be
63 # used. By including these in order from most machine dependent
64 # to most machine independent, we allow a machine dependent file
65 # to be used in preference over a machine independent version
66 # (Such as a machine specific optimization, which preserves the
67 # interfaces.)
68 #
69 include $(UTSTREE)/intel/Makefile.files
70 include $(UTSTREE)/common/Makefile.files
72 #
73 # ----- TRANSITIONAL SECTION -----
74 #
76 #
77 # Not everything which *should* be a module is a module yet. The
78 # following is a list of such objects which are currently part of
79 # genunix but which might someday become kmods. This must be
80 # defined before we include Makefile.uts, or else genunix's build
81 # won't be as parallel as we might like.
82 #
83 NOT_YET_KMODS = $(OLDPTY_OBJS) $(PTY_OBJS) $(VCONS_CONF_OBJS) $(MOD_OBJS)
85 #
86 # ----- END OF TRANSITIONAL SECTION -----
87 #
88 # Include machine independent rules. Note that this does not imply
89 # that the resulting module from rules in Makefile.uts is machine
90 # independent. Only that the build rules are machine independent.
91 #
92 include $(UTSBASE)/Makefile.uts
94 #
95 # The following must be defined for all implementations:
96 #
97 MODSTUBS = $(UTSBASE)/intel/ia32/ml/modstubs.s
99 #
100 # Define supported builds
101 #
102 DEF_BUILDS = $(DEF_BUILDS64) $(DEF_BUILDS32)
103 ALL_BUILDS = $(ALL_BUILDS64) $(ALL_BUILDS32)
105 #
106 # x86 or amd64 inline templates
107 #
108 INLINES_32 = $(UTSBASE)/intel/ia32/ml/ia32.il
109 INLINES_64 = $(UTSBASE)/intel/amd64/ml/amd64.il
110 INLINES += $(INLINES_$(CLASS))
112 #
113 # kernel-specific optimizations; override default in Makefile.master
114 #
116 CFLAGS_XARCH_32 = $(i386_CFLAGS)
117 CFLAGS_XARCH_64 = $(amd64_CFLAGS)
118 CFLAGS_XARCH = $(CFLAGS_XARCH_$(CLASS))
120 COPTFLAG_32 = $(COPTFLAG)
121 COPTFLAG_64 = $(COPTFLAG64)
122 COPTIMIZE = $(COPTFLAG_$(CLASS))
124 CFLAGS = $(CFLAGS_XARCH)
125 CFLAGS += $(COPTIMIZE)
126 CFLAGS += $(INLINES) -D_ASM_INLINES
127 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        += audiopl6x
207 DRV_KMODS        += audiopci
208 DRV_KMODS        += audiosolo
209 DRV_KMODS        += audiotst
210 DRV_KMODS        += audiovial823x
211 DRV_KMODS_32    += audiovial97
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        += myril0ge
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 #endif /* ! codereview */
235 DRV_KMODS        += devinfo
236 DRV_KMODS        += dld
237 DRV_KMODS        += dlpistub
238 DRV_KMODS_32    += dnet
239 DRV_KMODS        += dump
240 DRV_KMODS        += ecpp
241 DRV_KMODS        += emlxs
242 DRV_KMODS        += fd
243 DRV_KMODS        += fdc
244 DRV_KMODS        += fm
245 DRV_KMODS        += fssnap
246 DRV_KMODS        += hxge
247 DRV_KMODS        += i8042
248 DRV_KMODS        += i915
249 DRV_KMODS        += icmp
250 DRV_KMODS        += icmp6
251 DRV_KMODS        += intel_nb5000
252 DRV_KMODS        += intel_nhm
253 DRV_KMODS        += ip
254 DRV_KMODS        += ip6
255 DRV_KMODS        += ipf
256 DRV_KMODS        += ipnet
257 DRV_KMODS        += ippctl
258 DRV_KMODS        += ipsecah
259 DRV_KMODS        += ipsecesp

```



```

260 DRV_KMODS += ipw
261 DRV_KMODS += iwh
262 DRV_KMODS += iwi
263 DRV_KMODS += iwk
264 DRV_KMODS += iwpm
265 DRV_KMODS += iwscn
266 DRV_KMODS += kb8042
267 DRV_KMODS += keysock
268 DRV_KMODS += kssl
269 DRV_KMODS += kstat
270 DRV_KMODS += ksysms
271 DRV_KMODS += kmdb
272 DRV_KMODS += llcl
273 DRV_KMODS += lofi
274 DRV_KMODS += log
275 DRV_KMODS += logindmux
276 DRV_KMODS += mega_sas
277 DRV_KMODS += mc-amd
278 DRV_KMODS += mm
279 DRV_KMODS += mouse8042
280 DRV_KMODS += mpt_sas
281 DRV_KMODS += mr_sas
282 DRV_KMODS += mwl
283 DRV_KMODS += nca
284 DRV_KMODS += nsmb
285 DRV_KMODS += nulldriver
286 DRV_KMODS += nv_sata
287 DRV_KMODS += nxge
288 DRV_KMODS += oce
289 DRV_KMODS += openeep
290 DRV_KMODS += pci_pci
291 DRV_KMODS += pcic
292 DRV_KMODS += pcieb
293 DRV_KMODS += phymem
294 DRV_KMODS += pcan
295 DRV_KMODS += pcwl
296 DRV_KMODS += pit_beeper
297 DRV_KMODS += pm
298 DRV_KMODS += poll
299 DRV_KMODS += pool
300 DRV_KMODS += power
301 DRV_KMODS += pseudo
302 DRV_KMODS += ptc
303 DRV_KMODS += ptm
304 DRV_KMODS += pts
305 DRV_KMODS += ptsl
306 DRV_KMODS += qlge
307 DRV_KMODS += radeon
308 DRV_KMODS += ral
309 DRV_KMODS += ramdisk
310 DRV_KMODS += random
311 DRV_KMODS += rds
312 DRV_KMODS += rds3
313 DRV_KMODS += rpcib
314 DRV_KMODS += rsm
315 DRV_KMODS += rts
316 DRV_KMODS += rtw
317 DRV_KMODS += rum
318 DRV_KMODS += rwd
319 DRV_KMODS += rwn
320 DRV_KMODS += sad
321 DRV_KMODS += sd
322 DRV_KMODS += sdhost
323 DRV_KMODS += sgen
324 DRV_KMODS += si3124
325 DRV_KMODS += smbios

```

```

326 DRV_KMODS += softmac
327 DRV_KMODS += spdssock
328 DRV_KMODS += smpsrv
329 DRV_KMODS += smp
330 DRV_KMODS += sppp
331 DRV_KMODS += sppptun
332 DRV_KMODS += srpt
333 DRV_KMODS += st
334 DRV_KMODS += sy
335 DRV_KMODS += sysevent
336 DRV_KMODS += sysmsg
337 DRV_KMODS += tcp
338 DRV_KMODS += tcp6
339 DRV_KMODS += tl
340 DRV_KMODS += tnf
341 DRV_KMODS += tpm
342 DRV_KMODS += trill
343 DRV_KMODS += udp
344 DRV_KMODS += udp6
345 DRV_KMODS += ucode
346 DRV_KMODS += ural
347 DRV_KMODS += uath
348 DRV_KMODS += urtw
349 DRV_KMODS += vgatext
350 DRV_KMODS += heci
351 DRV_KMODS += vnic
352 DRV_KMODS += vscan
353 DRV_KMODS += wc
354 DRV_KMODS += winlock
355 DRV_KMODS += wpi
356 DRV_KMODS += xge
357 DRV_KMODS += yge
358 DRV_KMODS += zcons
359 DRV_KMODS += zyd
360 DRV_KMODS += simnet
361 DRV_KMODS += stmf
362 DRV_KMODS += stmf_sbd
363 DRV_KMODS += fct
364 DRV_KMODS += fcoe
365 DRV_KMODS += fcoet
366 DRV_KMODS += fcoei
367 DRV_KMODS += qlt
368 DRV_KMODS += iscsit
369 DRV_KMODS += pppt
370 DRV_KMODS += ncall nsctl sdbc nskern sv
371 DRV_KMODS += ii rdc rdcsrv rdcstub
372 DRV_KMODS += iptun

374 $(CLOSED_BUILD)CLOSED_DRV_KMODS += bmc
375 $(CLOSED_BUILD)CLOSED_DRV_KMODS += glm
376 $(CLOSED_BUILD)CLOSED_DRV_KMODS += intel_nhmex
377 $(CLOSED_BUILD)CLOSED_DRV_KMODS += cpqary3
378 $(CLOSED_BUILD)CLOSED_DRV_KMODS += marvell88sx
379 $(CLOSED_BUILD)CLOSED_DRV_KMODS += bcm_sata
380 $(CLOSED_BUILD)CLOSED_DRV_KMODS += memtest
381 $(CLOSED_BUILD)CLOSED_DRV_KMODS += mpt
382 $(CLOSED_BUILD)CLOSED_DRV_KMODS += atiatom
383 $(CLOSED_BUILD)CLOSED_DRV_KMODS += acpi_toshiba

385 #
386 # Common code drivers
387 #

389 DRV_KMODS += afe
390 DRV_KMODS += atge
391 DRV_KMODS += bfe

```

```

392 DRV_KMODS      += dmfe
393 DRV_KMODS      += e1000g
394 DRV_KMODS      += efe
395 DRV_KMODS      += elxl
396 DRV_KMODS      += hme
397 DRV_KMODS      += mxfe
398 DRV_KMODS      += nge
399 DRV_KMODS      += pcn
400 DRV_KMODS      += rge
401 DRV_KMODS      += rtls
402 DRV_KMODS      += sfe
403 DRV_KMODS      += amd8111s
404 DRV_KMODS      += igb
405 DRV_KMODS      += iprb
406 DRV_KMODS      += ixgbe
407 DRV_KMODS      += vr
408 $(CLOSED_BUILD)CLOSED_DRV_KMODS += ixgb

```

```

410 #
411 #      DTrace and DTrace Providers
412 #
413 DRV_KMODS      += dtrace
414 DRV_KMODS      += fbt
415 DRV_KMODS      += lockstat
416 DRV_KMODS      += profile
417 DRV_KMODS      += sdt
418 DRV_KMODS      += systrace
419 DRV_KMODS      += fasttrap
420 DRV_KMODS      += dcpc

```

```

422 #
423 #      I/O framework test drivers
424 #
425 DRV_KMODS      += pshot
426 DRV_KMODS      += gen_drv
427 DRV_KMODS      += tvhci tphci tclient
428 DRV_KMODS      += emul64

```

```

430 #
431 #      Machine Specific Driver Modules (/kernel/drv):
432 #
433 DRV_KMODS      += options
434 DRV_KMODS      += scsi_vhci
435 DRV_KMODS      += pmcs
436 DRV_KMODS      += pmcs8001fw
437 DRV_KMODS      += arcmsr
438 DRV_KMODS      += fcp
439 DRV_KMODS      += fcip
440 DRV_KMODS      += fcsn
441 DRV_KMODS      += fp
442 DRV_KMODS      += qlc
443 DRV_KMODS      += iscsi

```

```

445 #
446 #      PCMCIA specific module(s)
447 #
448 DRV_KMODS      += pcs
449 DRV_KMODS      += pcata
450 MISC_KMODS     += cardbus
451 $(CLOSED_BUILD)CLOSED_DRV_KMODS += pcser

```

```

453 #
454 #      SCSI Enclosure Services driver
455 #
456 DRV_KMODS      += ses

```

```

458 #
459 #      USB specific modules
460 #
461 DRV_KMODS      += hid
462 DRV_KMODS      += hwarc hwahc
463 DRV_KMODS      += hubd
464 DRV_KMODS      += uhci
465 DRV_KMODS      += ehci
466 DRV_KMODS      += ohci
467 DRV_KMODS      += usb_mid
468 DRV_KMODS      += usb_ia
469 DRV_KMODS      += scsa2usb
470 DRV_KMODS      += usbprn
471 DRV_KMODS      += ugen
472 DRV_KMODS      += usbser
473 DRV_KMODS      += usbsacm
474 DRV_KMODS      += usbsksp
475 DRV_KMODS      += usbsprl
476 DRV_KMODS      += usb_ac
477 DRV_KMODS      += usb_as
478 DRV_KMODS      += usbskel
479 DRV_KMODS      += usbvc
480 DRV_KMODS      += usbftdi
481 DRV_KMODS      += wusb_df
482 DRV_KMODS      += wusb_ca
483 DRV_KMODS      += usbecm

```

```

485 $(CLOSED_BUILD)CLOSED_DRV_KMODS += usbser_edge

```

```

487 #
488 #      1394 modules
489 #
490 MISC_KMODS     += s1394 sbp2
491 DRV_KMODS      += hci1394 scsa1394
492 DRV_KMODS      += av1394
493 DRV_KMODS      += dcam1394

```

```

495 #
496 #      InfiniBand pseudo drivers
497 #
498 DRV_KMODS      += ib ibp eibnx eoib rdsib sdp iser daplt hermon tavor sol_ucma
499 DRV_KMODS      += sol_umad

```

```

501 #
502 #      LVM modules
503 #
504 DRV_KMODS      += md
505 MISC_KMODS     += md_stripe md_hotspares md_mirror md_raid md_trans md_notify
506 MISC_KMODS     += md_sp

```

```

508 #
509 #      Brand modules
510 #
511 BRAND_KMODS    += snl_brand s10_brand

```

```

513 #
514 #      Exec Class Modules (/kernel/exec):
515 #
516 EXEC_KMODS     += elfexec intpexec shbinexec javaexec

```

```

518 #
519 #      Scheduling Class Modules (/kernel/sched):
520 #
521 SCHED_KMODS    += IA RT TS RT_DPTBL TS_DPTBL FSS FX FX_DPTBL SDC

```

```

523 #

```

```

524 #       File System Modules (/kernel/fs):
525 #
526 FS_KMODS      += autofscs cachefs ctfs dcfs dev devfs fdcs fifofs hsfs lofs
527 FS_KMODS      += mntfs namefs nfs objfs zfs zut
528 FS_KMODS      += pcfs procfs sockfs specfs tmpfs udfs ufs sharefs
529 FS_KMODS      += smbfs

531 #
532 #       Streams Modules (/kernel/strmod):
533 #
534 STRMOD_KMODS  += bufmod connld dedump ldterm pckt pfmod pipemod
535 STRMOD_KMODS  += ptem redirmoed rpcmod rlmod telmod timod
536 STRMOD_KMODS  += sppedasyn spppcomp
537 STRMOD_KMODS  += tirdwr ttcompat
538 STRMOD_KMODS  += usbkbm
539 STRMOD_KMODS  += usbms
540 STRMOD_KMODS  += usbwcm
541 STRMOD_KMODS  += usb_ah
542 STRMOD_KMODS  += dircompat
543 STRMOD_KMODS  += cryptomod
544 STRMOD_KMODS  += vuid2ps2
545 STRMOD_KMODS  += vuid3ps2
546 STRMOD_KMODS  += vuidm3p
547 STRMOD_KMODS  += vuidm4p
548 STRMOD_KMODS  += vuidm5p

550 #
551 #       'System' Modules (/kernel/sys):
552 #
553 SYS_KMODS     += c2audit
554 SYS_KMODS     += doorfs
555 SYS_KMODS     += exacctsys
556 SYS_KMODS     += inst_sync
557 SYS_KMODS     += kaio
558 SYS_KMODS     += msgsys
559 SYS_KMODS     += pipe
560 SYS_KMODS     += portfs
561 SYS_KMODS     += pset
562 SYS_KMODS     += semsys
563 SYS_KMODS     += shmsys
564 SYS_KMODS     += sysacct
565 SYS_KMODS     += acctctl

567 #
568 #       'Misc' Modules (/kernel/misc)
569 #       MISC_KMODS are built both 32-bit and 64-bit
570 #       MISC_KMODS_32 are built only 32-bit
571 #       MISC_KMODS_64 are built only 64-bit
572 #
573 MISC_KMODS    += ac97
574 MISC_KMODS    += acpica
575 MISC_KMODS    += agpmaster
576 MISC_KMODS    += bignum
577 MISC_KMODS    += bootdev
578 MISC_KMODS    += busra
579 MISC_KMODS    += cmlb
580 MISC_KMODS    += consconfig
581 MISC_KMODS    += ctf
582 MISC_KMODS    += dadk
583 MISC_KMODS    += dcopy
584 MISC_KMODS    += dls
585 MISC_KMODS    += drm
586 MISC_KMODS    += fssnap_if
587 MISC_KMODS    += gda
588 MISC_KMODS    += gld
589 MISC_KMODS    += hidparser

```

```

590 MISC_KMODS    += hook
591 MISC_KMODS    += hpcsvc
592 MISC_KMODS    += ibcm
593 MISC_KMODS    += ibdm
594 MISC_KMODS    += ibdma
595 MISC_KMODS    += ibmf
596 MISC_KMODS    += ibtl
597 MISC_KMODS    += idm
598 MISC_KMODS    += idmap
599 MISC_KMODS    += iomulib
600 MISC_KMODS    += ipc
601 MISC_KMODS    += kbtrans
602 MISC_KMODS    += kcf
603 MISC_KMODS    += kgssapi
604 MISC_KMODS    += kmecch_dummy
605 MISC_KMODS    += kmecch_krb5
606 MISC_KMODS    += ksocket
607 MISC_KMODS    += mac
608 MISC_KMODS    += mii
609 MISC_KMODS    += mwlfw
610 MISC_KMODS    += net80211
611 MISC_KMODS    += nfs_dlboot
612 MISC_KMODS    += nfssrv
613 MISC_KMODS    += neti
614 MISC_KMODS    += pci_autoconfig
615 MISC_KMODS    += pcicfg
616 MISC_KMODS    += pcihp
617 MISC_KMODS    += pcmcia
618 MISC_KMODS    += rpcsec
619 MISC_KMODS    += rpcsec_gss
620 MISC_KMODS    += rsmops
621 MISC_KMODS    += sata
622 MISC_KMODS    += scsi
623 MISC_KMODS    += sda
624 MISC_KMODS    += sol_ofs
625 MISC_KMODS    += spuni
626 MISC_KMODS    += strategy
627 MISC_KMODS    += strplumb
628 MISC_KMODS    += tem
629 MISC_KMODS    += tlimod
630 MISC_KMODS    += usba usba10 usbs49_fw
631 MISC_KMODS    += scsi_vhci_f_sym_hds
632 MISC_KMODS    += scsi_vhci_f_sym
633 MISC_KMODS    += scsi_vhci_f_tpgs
634 MISC_KMODS    += scsi_vhci_f_asym_sun
635 MISC_KMODS    += scsi_vhci_f_tape
636 MISC_KMODS    += scsi_vhci_f_tpgs_tape
637 MISC_KMODS    += fctl
638 MISC_KMODS    += emlxs_fw
639 MISC_KMODS    += qlc_fw_2200
640 MISC_KMODS    += qlc_fw_2300
641 MISC_KMODS    += qlc_fw_2400
642 MISC_KMODS    += qlc_fw_2500
643 MISC_KMODS    += qlc_fw_6322
644 MISC_KMODS    += qlc_fw_8100
645 MISC_KMODS    += hwa1480_fw
646 MISC_KMODS    += uathfw
647 MISC_KMODS    += uwba

649 $(CLOSED_BUILD)CLOSED_MISC_KMODS    += klmmod klmops
650 $(CLOSED_BUILD)CLOSED_MISC_KMODS    += scsi_vhci_f_asym_lsi
651 $(CLOSED_BUILD)CLOSED_MISC_KMODS    += scsi_vhci_f_asym_emc
652 $(CLOSED_BUILD)CLOSED_MISC_KMODS    += scsi_vhci_f_sym_emc

654 #
655 #       Software Cryptographic Providers (/kernel/crypto):

```

```

656 #
657 CRYPTO_KMODS += aes
658 CRYPTO_KMODS += arcfour
659 CRYPTO_KMODS += blowfish
660 CRYPTO_KMODS += des
661 CRYPTO_KMODS += ecc
662 CRYPTO_KMODS += md4
663 CRYPTO_KMODS += md5
664 CRYPTO_KMODS += rsa
665 CRYPTO_KMODS += sha1
666 CRYPTO_KMODS += sha2
667 CRYPTO_KMODS += swrand

669 #
670 #     IP Policy Modules (/kernel/ipp)
671 #
672 IPP_KMODS += dlcosmck
673 IPP_KMODS += flowacct
674 IPP_KMODS += ipgpc
675 IPP_KMODS += dscpmk
676 IPP_KMODS += tokenmt
677 IPP_KMODS += tswtclmt

679 #
680 #     generic-unix module (/kernel/genunix):
681 #
682 GENUNIX_KMODS += genunix

684 #
685 #     SVVS Testing Modules (/kernel/strmod):
686 #
687 #     These are streams and driver modules which are not to be
688 #     delivered with a released system. However, during development
689 #     it is convenient to build and install the SVVS kernel modules.
690 #
691 SVVS_KMODS += lmodb lmode lmodr lmodt svvslo tidg tivc tmux

693 $(CLOSED_BUILD)SVVS += svvs

695 #
696 #     Modules eXcluded from the product:
697 #
698 $(CLOSED_BUILD)CLOSED_XMODS = \
699     adpu320 \
700     bnx \
701     bnxe \
702     lsimega \
703     sdpiib

706 #
707 #     'Dacf' Modules (/kernel/dacf):
708 #

710 #
711 #     Performance Counter BackEnd modules (/usr/kernel/pcbe)
712 #
713 PCBE_KMODS += p123_pcbe p4_pcbe opteron_pcbe core_pcbe

715 #
716 #     MAC-Type Plugin Modules (/kernel/mac)
717 #
718 MAC_KMODS += mac_6to4
719 MAC_KMODS += mac_ether
720 MAC_KMODS += mac_ipv4
721 MAC_KMODS += mac_ipv6

```

```

722 MAC_KMODS += mac_wifi
723 MAC_KMODS += mac_ib

725 #
726 #     socketmod (kernel/socketmod)
727 #
728 SOCKET_KMODS += sockpfp
729 SOCKET_KMODS += socksctp
730 SOCKET_KMODS += socksdp
731 SOCKET_KMODS += socktrds
732 SOCKET_KMODS += ksslif

734 #
735 #     kiconv modules (/kernel/kiconv):
736 #
737 KICONV_KMODS += kiconv_emea kiconv_ja kiconv_ko kiconv_sc kiconv_tc

739 #
740 #     'Dacf' Modules (/kernel/dacf):
741 #
742 DACF_KMODS += net_dacf

```

2348 Wed Jun 13 12:05:44 2012

new/usr/src/uts/intel/dccp/Makefile

%B

```

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 #
21 #
22 # uts/intel/dccp/Makefile
23 #
24 # Copyright 2008 Sun Microsystems, Inc. All rights reserved.
25 # Use is subject to license terms.
26 #
27 # This makefile drives the production of the dccp driver kernel module.
28 #
29 # intel implementation architecture dependent
30 #
31 #
32 #
33 # Path to the base of the uts directory tree (usually /usr/src/uts).
34 #
35 UTSBASE = ../..
36 #
37 #
38 # Define the module and object file sets.
39 #
40 MODULE = dccp
41 OBJECTS = $(DCCP_OBJS:%=$(OBJS_DIR)/%)
42 LINTS = $(DCCP_OBJS:%.o=$(LINTS_DIR)/%.ln)
43 ROOTMODULE = $(ROOT_DRV_DIR)/$(MODULE)
44 ROOTLINK = $(ROOT_STRMOD_DIR)/$(MODULE) $(ROOT SOCK_DIR)/$(MODULE)
45 CONF_SRCDIR = $(UTSBASE)/common/inet/dccp
46 #
47 #
48 # Extra for $(MODULE).check target
49 #
50 # Need to remove ipddi.o since it has non-static defines for _init etc.
51 IP_CHECK_OBJS = $(IP_OBJS:ipddi.o=ip.o)
52 EXTRA_CHECK_OBJS = $(IP_CHECK_OBJS:%=../ip/$(OBJS_DIR)/%)
53 #
54 #
55 # Include common rules.
56 #
57 include $(UTSBASE)/intel/Makefile.intel
58 #
59 #
60 # Define targets
61 #

```

```

62 ALL_TARGET = $(BINARY) $(SRC_CONFILE)
63 LINT_TARGET = $(MODULE).lint
64 INSTALL_TARGET = $(BINARY) $(ROOTMODULE) $(ROOTLINK) $(ROOT_CONFFILE)
65 #
66 #
67 # depends on ip and sockfs
68 #
69 LDFLAGS += -dy -Ndrv/ip -Nfs/sockfs
70 #
71 #
72 # Default build targets.
73 #
74 .KEEP_STATE:
75 #
76 def: $(DEF_DEPS)
77 #
78 all: $(ALL_DEPS)
79 #
80 clean: $(CLEAN_DEPS)
81 #
82 clobber: $(CLOBBER_DEPS)
83 #
84 lint: $(LINT_DEPS)
85 #
86 modlintlib: $(MODLINTLIB_DEPS)
87 #
88 clean.lint: $(CLEAN_LINT_DEPS)
89 #
90 install: $(INSTALL_DEPS)
91 #
92 $(ROOTLINK): $(ROOT_STRMOD_DIR) $(ROOT SOCK_DIR) $(ROOTMODULE)
93 -$(RM) $@; ln $(ROOTMODULE) $@
94 #
95 #
96 # Include common targets.
97 #
98 include $(UTSBASE)/intel/Makefile.targ
99 #endif /* ! codereview */

```

```

*****
2178 Wed Jun 13 12:05:44 2012
new/usr/src/uts/intel/dccp6/Makefile
%B
*****
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 2004 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #
26 # This makefile drives the production of the dccp6 driver kernel module.
27 #
28 # intel implementation architecture dependent
29 #
30 #
31 #
32 # Path to the base of the uts directory tree (usually /usr/src/uts).
33 #
34 UTSBASE = ../../

36 #
37 # Define the module and object file sets.
38 #
39 MODULE = dccp6
40 OBJECTS = $(TCP6_OBJS:%=$(OBJS_DIR)/%)
41 LINTS = $(TCP6_OBJS:%.o=$(LINTS_DIR)/%.ln)
42 ROOTMODULE = $(ROOT_DRV_DIR)/$(MODULE)
43 CONF_SRCDIR = $(UTSBASE)/common/inet/dccp

45 #
46 # Extra for $(MODULE).check target
47 #
48 # Need to remove ipddi.o since it has non-static defines for _init etc.
49 IP_CHECK_OBJS = $(IP_OBJS:ipddi.o=ip.o)
50 EXTRA_CHECK_OBJS = $(IP_CHECK_OBJS:%=../ip/$(OBJS_DIR)/%)

52 #
53 # Include common rules.
54 #
55 include $(UTSBASE)/intel/Makefile.intel

57 #
58 # Define targets
59 #
60 ALL_TARGET = $(BINARY) $(SRC_CONFFILE)
61 LINT_TARGET = $(MODULE).lint

```

```

62 INSTALL_TARGET = $(BINARY) $(ROOTMODULE) $(ROOT_CONFFILE)

64 #
65 # depends on tcp ip and ip6
66 #
67 LDFLAGS += -dy -Ndrv/tcp -Ndrv/ip -Ndrv/ip6

69 #
70 # Default build targets.
71 #
72 .KEEP_STATE:

74 def: $(DEF_DEPS)

76 all: $(ALL_DEPS)

78 clean: $(CLEAN_DEPS)

80 clobber: $(CLOBBER_DEPS)

82 lint: $(LINT_DEPS)

84 modlintlib: $(MODLINTLIB_DEPS)

86 clean.lint: $(CLEAN_LINT_DEPS)

88 install: $(INSTALL_DEPS)

90 #
91 # Include common targets.
92 #
93 include $(UTSBASE)/intel/Makefile.targ
94 #endif /* ! codereview */

```

new/usr/src/uts/intel/ip/ip.global-objs.debug64

1

6046 Wed Jun 13 12:05:45 2012

new/usr/src/uts/intel/ip/ip.global-objs.debug64

%B

```
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 #
21 #
22 # Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
23 # Copyright 2011 Nexenta Systems, Inc. All rights reserved
24 #
```

```
26 arp_m_tbl
27 arp_mod_info
28 arp_netinfo
29 arp_no_defense
30 arpinfo
31 cb_inet_devops
32 cl_inet_bind
33 cl_inet_checkspi
34 cl_inet_connect2
35 cl_inet_deletespi
36 cl_inet_disconnect
37 cl_inet_getspi
38 cl_inet_idlesa
39 cl_inet_ipident
40 cl_inet_isclusterwide
41 cl_inet_listen
42 cl_inet_unbind
43 cl_inet_unlisten
44 cl_sctp_assoc_change
45 cl_sctp_check_addr
46 cl_sctp_connect
47 cl_sctp_disconnect
48 cl_sctp_listen
49 cl_sctp_unlisten
50 conn_drain_nthreads
51 dccp_bind_fanout_size
52 dccp_conn_cache
53 dccp_max_optsize
54 dccp_opt_arr
55 dccp_opt_obj
56 dccp_propinfo_count
57 dccp_propinfo_tbl
58 dccp_random_anon_port
59 dccp_rinfo
60 dccp_rinitv4
61 dccp_rinitv6
```

new/usr/src/uts/intel/ip/ip.global-objs.debug64

2

```
62 dccp_valid_levels_arr
63 dccp_winfo
64 dccp_winit
65 dccpinfov4
66 dccpinfov6
67 #endif /* ! codereview */
68 dce_cache
69 default_ip6_asp_table
70 do_tcp_fusion
71 do_tcpzcopy
72 dohwcksum
73 dummy_mod_info
74 dummymodinfo
75 dummyrmodinit
76 dummywmodinit
77 eventq_queue_in
78 eventq_queue_nic
79 eventq_queue_out
80 fsw
81 gcdb_hash
82 gcdb_hash_size
83 gcdb_lock
84 gcgrp4_hash
85 gcgrp6_hash
86 gcgrp_hash_size
87 gcgrp_lock
88 icmp_fallback_sock_winit
89 icmp_frag_size_table
90 icmp_g_t_info_ack
91 icmp_ipha
92 icmp_max_optsize
93 icmp_mod_info
94 icmp_opt_arr
95 icmp_opt_obj
96 icmp_propinfo_tbl
97 icmp_valid_levels_arr
98 icmpinfov4
99 icmpinfov6
100 icmprinitv4
101 icmprinitv6
102 icmpwinit
103 ilb_conn_cache
104 ilb_conn_cache_timeout
105 ilb_conn_hash_size
106 ilb_conn_tcp_expiry
107 ilb_conn_timer_size
108 ilb_conn_udp_expiry
109 ilb_kstat_instance
110 ilb_kmem_flags
111 ilb_nat_src_hash_size
112 ilb_nat_src_instance
113 ilb_rule_hash_size
114 ilb_sticky_cache
115 ilb_sticky_hash_size
116 ilb_sticky_expiry
117 ilb_sticky_timer_size
118 ilb_sticky_timeout
119 ill_no_arena
120 ill_null
121 inet_dev_info
122 inet_devops
123 ip6_fhtable_hash_size
124 ip6opt_ls
125 ip_cgtp_filter_rev
126 ip_conn_cache
127 ip_debug
```

```

128 ip_g_all_ones
129 ip_helper_stream_info
130 ip_helper_stream_rinit
131 ip_helper_stream_winit
132 ip_ioctl_ftbl
133 ip_loopback_mtu_v6plus
134 ip_loopback_mtuplus
135 ip_m_tbl
136 ip_max_frag_dups
137 ip_min_frag_prune_time
138 ip_minor_arena_la
139 ip_minor_arena_sa
140 ip_misc_ioctl_count
141 ip_misc_ioctl_table
142 ip_mod_info
143 ip_modclose_ackwait_ms
144 ip_ndx_ioctl_count
145 ip_ndx_ioctl_table
146 ip_poll_normal_ms
147 ip_poll_normal_ticks
148 ip_propinfo_tbl
149 ip_propinfo_count
150 ip_rput_pullups
151 ip_six_byte_all_ones
152 ip_squeue_create_callback
153 ip_squeue_enter
154 ip_squeue_fanout
155 ip_squeue_flag
156 ip_squeue_worker_wait
157 ip_thread_data
158 ip_thread_list
159 ip_thread_rwlock
160 ipcl_bind_fanout_size
161 ipcl_conn_hash_maxsize
162 ipcl_conn_hash_memfactor
163 ipcl_conn_hash_size
164 ipcl_dccp_fanout_size
165 #endif /* ! codereview */
166 ipcl iptun_fanout_size
167 ipcl_raw_fanout_size
168 ipcl_udp_fanout_size
169 ipif_loopback_name
170 ipif_zero
171 ipinfov4
172 ipinfov6
173 iplrinit
174 iplwinit
175 ipmp_kstats
176 iprinitv4
177 iprinitv6
178 ipsec_action_cache
179 ipsec_hdr_pullup_needed
180 ipsec_pol_cache
181 ipsec_policy_failure_msgs
182 ipsec_sel_cache
183 ipsec_spd_hashsize
184 ipsec_weird_null_inbound_policy
185 ipv4info
186 ipv6_all_hosts_mcast
187 ipv6_all_ones
188 ipv6_all_rtrs_mcast
189 ipv6_all_v2rtrs_mcast
190 ipv6_all_zeros
191 ipv6_ll_template
192 ipv6_loopback
193 ipv6_solicited_node_mcast

```

```

194 ipv6_unspecified_group
195 ipv6info
196 ipwinit
197 ire_cache
198 ire_gw_secattr_cache
199 ire_null
200 ire_nv_arr
201 ire_nv_tbl
202 lcl_param_arr
203 mask_rnhead
204 max_keylen
205 modldrv
206 modlinkage
207 modlstrmod
208 multicast_encap_iphdr
209 nce_cache
210 ncec_cache
211 netdev_privs
212 prov_update_handle
213 radix_mask_cache
214 radix_node_cache
215 rawip_conn_cache
216 recvq_call
217 recvq_loop_cnt
218 req_arr
219 rinit_arp
220 rn_mkfreelist
221 rn_ones
222 rn_zeros
223 rt_entry_cache
224 rts_conn_cache
225 rts_g_t_info_ack
226 rts_max_optsize
227 rts_mod_info
228 rts_opt_arr
229 rts_opt_obj
230 rts_valid_levels_arr
231 rtsinfo
232 rtsrinit
233 rtswinit
234 sctp_asconf_default_dispatch
235 sctp_asconf_dispatch_tbl
236 sctp_conn_cache
237 sctp_conn_hash_size
238 sctp_do_reclaim
239 sctp_kmem_faddr_cache
240 sctp_kmem_ftsn_set_cache
241 sctp_kmem_set_cache
242 sctp_min_assoc_listener
243 sctp_opt_arr
244 sctp_opt_arr_size
245 sctp_pa_early_abort
246 sctp_pp_early_abort
247 sctp_propinfo_tbl
248 sctp_propinfo_count
249 sctp_recvq_tq_list_max
250 sctp_recvq_tq_task_min
251 sctp_recvq_tq_thr_max
252 sctp_recvq_tq_thr_min
253 sctp_sin6_null
254 sctpdebug
255 sin6_null
256 sin_null
257 skip_sctp_cksum
258 sock_dccp_downcalls
259 #endif /* ! codereview */

```



```
260 sock_rawip_downcalls
261 sock_rts_downcalls
262 sock_tcp_downcalls
263 sock_udp_downcalls
264 sqset_global_list
265 sqset_global_size
266 sqset_lock
267 squeue_cache
268 squeue_drain_ms
269 squeue_drain_ns
270 squeue_workerwait_ms
271 squeue_workerwait_tick
272 tcp_acceptor_rinit
273 tcp_acceptor_winit
274 tcp_conn_cache
275 tcp_conn_hash_size
276 tcp_do_reclaim
277 tcp_drop_ack_unsent_cnt
278 tcp_dummy_upcalls
279 tcp_early_abort
280 tcp_fallback_sock_winit
281 tcp_free_list_max_cnt
282 tcp_g_kstat
283 tcp_g_statistics
284 tcp_g_t_info_ack
285 tcp_g_t_info_ack_v6
286 tcp_icmp_source_quench
287 tcp_init_wnd_chk
288 tcp_max_init_cwnd
289 tcp_max_optsize
290 tcp_min_conn_listener
291 tcp_notsack_blk_cache
292 tcp_opt_arr
293 tcp_opt_obj
294 tcp_outbound_squeue_switch
295 tcp_propinfo_tbl
296 tcp_propinfo_count
297 tcp_random_anon_port
298 tcp_random_end_ptr
299 tcp_random_fptr
300 tcp_random_lock
301 tcp_random_rptr
302 tcp_random_state
303 tcp_randtbl
304 tcp_rinfo
305 tcp_rinitv4
306 tcp_rinitv6
307 tcp_sock_winit
308 tcp_squeue_flag
309 tcp_squeue_wput
310 tcp_static_maxpsz
311 tcp_timercache
312 tcp_tx_pull_len
313 tcp_valid_levels_arr
314 tcp_winfo
315 tcp_winit
316 tcpinfov4
317 tcpinfov6
318 tli_errs
319 tsol_strict_error
320 tun_spd_hashsize
321 udp_bind_fanout_size
322 udp_conn_cache
323 udp_fallback_sock_winit
324 udp_g_t_info_ack_ipv4
325 udp_g_t_info_ack_ipv6
```

```
326 udp_lrinit
327 udp_lwinit
328 udp_max_optsize
329 udp_mod_info
330 udp_opt_arr
331 udp_opt_obj
332 udp_propinfo_tbl
333 udp_propinfo_count
334 udp_random_anon_port
335 udp_rinitv4
336 udp_rinitv6
337 udp_valid_levels_arr
338 udp_winit
339 udpinfov4
340 udpinfov6
341 winit_arp
342 nxge_cksum_workaround
```

6004 Wed Jun 13 12:05:47 2012

%B

```

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 #
21 #
22 # Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
23 # Copyright 2011 Nexenta Systems, Inc. All rights reserved
24 #

```

```

26 arp_m_tbl
27 arp_mod_info
28 arp_netinfo
29 arp_no_defense
30 arpinfo
31 cb_inet_devops
32 cl_inet_bind
33 cl_inet_checkspi
34 cl_inet_connect2
35 cl_inet_deletespi
36 cl_inet_disconnect
37 cl_inet_getspi
38 cl_inet_idlesa
39 cl_inet_ipident
40 cl_inet_isclusterwide
41 cl_inet_listen
42 cl_inet_unbind
43 cl_inet_unlisten
44 cl_sctp_assoc_change
45 cl_sctp_check_addr
46 cl_sctp_connect
47 cl_sctp_disconnect
48 cl_sctp_listen
49 cl_sctp_unlisten
50 conn_drain_nthreads
51 dccp_bind_fanout_size
52 dccp_conn_cache
53 dccp_max_optsize
54 dccp_opt_arr
55 dccp_opt_obj
56 dccp_propinfo_count
57 dccp_propinfo_tbl
58 dccp_random_anon_port
59 dccp_rinfo
60 dccp_rinitv4
61 dccp_rinitv6

```

```

62 dccp_valid_levels_arr
63 dccp_winfo
64 dccp_winit
65 dccpinfov4
66 dccpinfov6
67 #endif /* ! codereview */
68 dce_cache
69 default_ip6_asp_table
70 do_tcp_fusion
71 do_tcpzcopy
72 dohwcksum
73 dummy_mod_info
74 dummymodinfo
75 dummyrmodinit
76 dummywmodinit
77 eventq_queue_in
78 eventq_queue_nic
79 eventq_queue_out
80 fsw
81 gcdb_hash
82 gcdb_hash_size
83 gcdb_lock
84 gcgrp4_hash
85 gcgrp6_hash
86 gcgrp_hash_size
87 gcgrp_lock
88 icmp_fallback_sock_winit
89 icmp_frag_size_table
90 icmp_g_t_info_ack
91 icmp_ipha
92 icmp_max_optsize
93 icmp_mod_info
94 icmp_opt_arr
95 icmp_opt_obj
96 icmp_propinfo_tbl
97 icmp_valid_levels_arr
98 icmpinfov4
99 icmpinfov6
100 icmprinitv4
101 icmprinitv6
102 icmpwinit
103 ilb_conn_cache
104 ilb_conn_cache_timeout
105 ilb_conn_hash_size
106 ilb_conn_tcp_expiry
107 ilb_conn_timer_size
108 ilb_conn_udp_expiry
109 ilb_kstat_instance
110 ilb_kmem_flags
111 ilb_nat_src_hash_size
112 ilb_nat_src_instance
113 ilb_rule_hash_size
114 ilb_sticky_cache
115 ilb_sticky_hash_size
116 ilb_sticky_expiry
117 ilb_sticky_timer_size
118 ilb_sticky_timeout
119 ill_no_arena
120 ill_null
121 inet_dev_info
122 inet_devops
123 ip6_fhtable_hash_size
124 ip6opt_ls
125 ip_cgtp_filter_rev
126 ip_conn_cache
127 ip_debug

```

```

128 ip_g_all_ones
129 ip_helper_stream_info
130 ip_helper_stream_rinit
131 ip_helper_stream_winit
132 ip_ioctl_ftbl
133 ip_loopback_mtu_v6plus
134 ip_loopback_mtuplus
135 ip_m_tbl
136 ip_max_frag_dups
137 ip_min_frag_prune_time
138 ip_minor_arena_la
139 ip_minor_arena_sa
140 ip_misc_ioctl_count
141 ip_misc_ioctl_table
142 ip_mod_info
143 ip_modclose_ackwait_ms
144 ip_ndx_ioctl_count
145 ip_ndx_ioctl_table
146 ip_poll_normal_ms
147 ip_poll_normal_ticks
148 ip_propinfo_tbl
149 ip_propinfo_count
150 ip_rput_pullups
151 ip_six_byte_all_ones
152 ip_squeue_create_callback
153 ip_squeue_enter
154 ip_squeue_fanout
155 ip_squeue_flag
156 ip_squeue_worker_wait
157 ip_thread_data
158 ip_thread_list
159 ip_thread_rwlock
160 ipcl_bind_fanout_size
161 ipcl_conn_hash_maxsize
162 ipcl_conn_hash_memfactor
163 ipcl_conn_hash_size
164 ipcl_dccp_fanout_size
165 #endif /* ! codereview */
166 ipcl iptun_fanout_size
167 ipcl_raw_fanout_size
168 ipcl_udp_fanout_size
169 ipif_loopback_name
170 ipif_zero
171 ipinfov4
172 ipinfov6
173 iplrinit
174 iplwinit
175 ipmp_kstats
176 iprinitv4
177 iprinitv6
178 ipsec_action_cache
179 ipsec_hdr_pullup_needed
180 ipsec_pol_cache
181 ipsec_policy_failure_msgs
182 ipsec_sel_cache
183 ipsec_spd_hashsize
184 ipsec_weird_null_inbound_policy
185 ipv4info
186 ipv6_all_hosts_mcast
187 ipv6_all_ones
188 ipv6_all_rtrs_mcast
189 ipv6_all_v2rtrs_mcast
190 ipv6_all_zeros
191 ipv6_ll_template
192 ipv6_loopback
193 ipv6_solicited_node_mcast

```

```

194 ipv6_unspecified_group
195 ipv6info
196 ipwinit
197 ire_cache
198 ire_gw_secattr_cache
199 ire_null
200 ire_nv_arr
201 ire_nv_tbl
202 lcl_param_arr
203 mask_rnhead
204 max_keylen
205 modldrv
206 modlinkage
207 modlstrmod
208 multicast_encap_iphdr
209 nce_cache
210 ncec_cache
211 netdev_privs
212 prov_update_handle
213 radix_mask_cache
214 radix_node_cache
215 rawip_conn_cache
216 req_arr
217 rinit_arp
218 rn_mkfreelist
219 rn_ones
220 rn_zeros
221 rt_entry_cache
222 rts_conn_cache
223 rts_g_t_info_ack
224 rts_max_optsize
225 rts_mod_info
226 rts_opt_arr
227 rts_opt_obj
228 rts_valid_levels_arr
229 rtsinfo
230 rtsrinit
231 rtswinit
232 sctp_asconf_default_dispatch
233 sctp_asconf_dispatch_tbl
234 sctp_conn_cache
235 sctp_conn_hash_size
236 sctp_do_reclaim
237 sctp_kmem_faddr_cache
238 sctp_kmem_ftsn_set_cache
239 sctp_kmem_set_cache
240 sctp_min_assoc_listener
241 sctp_opt_arr
242 sctp_opt_arr_size
243 sctp_pa_early_abort
244 sctp_pp_early_abort
245 sctp_propinfo_tbl
246 sctp_propinfo_count
247 sctp_recvq_tq_list_max
248 sctp_recvq_tq_task_min
249 sctp_recvq_tq_thr_max
250 sctp_recvq_tq_thr_min
251 sctp_sin6_null
252 sctpdebug
253 sin6_null
254 sin_null
255 sock_dccp_downcalls
256 #endif /* ! codereview */
257 sock_rawip_downcalls
258 sock_rts_downcalls
259 sock_tcp_downcalls

```

```
260 sock_udp_downcalls
261 sqset_global_list
262 sqset_global_size
263 sqset_lock
264 squeue_cache
265 squeue_drain_ms
266 squeue_drain_ns
267 squeue_workerwait_ms
268 squeue_workerwait_tick
269 tcp_acceptor_rinit
270 tcp_acceptor_winit
271 tcp_conn_cache
272 tcp_conn_hash_size
273 tcp_do_reclaim
274 tcp_drop_ack_unsent_cnt
275 tcp_dummy_upcalls
276 tcp_early_abort
277 tcp_fallback_sock_winit
278 tcp_free_list_max_cnt
279 tcp_g_kstat
280 tcp_g_statistics
281 tcp_g_t_info_ack
282 tcp_g_t_info_ack_v6
283 tcp_icmp_source_quench
284 tcp_init_wnd_chk
285 tcp_max_init_cwnd
286 tcp_max_optsize
287 tcp_min_conn_listener
288 tcp_notsack_blk_cache
289 tcp_opt_arr
290 tcp_opt_obj
291 tcp_outbound_squeue_switch
292 tcp_propinfo_tbl
293 tcp_propinfo_count
294 tcp_random_anon_port
295 tcp_random_end_ptr
296 tcp_random_fptr
297 tcp_random_lock
298 tcp_random_rptr
299 tcp_random_state
300 tcp_randtbl
301 tcp_rinfo
302 tcp_rinitv4
303 tcp_rinitv6
304 tcp_sock_winit
305 tcp_squeue_flag
306 tcp_squeue_wput
307 tcp_static_maxpsz
308 tcp_timercache
309 tcp_tx_pull_len
310 tcp_valid_levels_arr
311 tcp_winfo
312 tcp_winit
313 tcpinfov4
314 tcpinfov6
315 tli_errs
316 tsol_strict_error
317 tun_spd_hashsize
318 udp_bind_fanout_size
319 udp_conn_cache
320 udp_fallback_sock_winit
321 udp_g_t_info_ack_ipv4
322 udp_g_t_info_ack_ipv6
323 udp_lrinit
324 udp_lwinit
325 udp_max_optsize
```

```
326 udp_mod_info
327 udp_opt_arr
328 udp_opt_obj
329 udp_propinfo_tbl
330 udp_propinfo_count
331 udp_random_anon_port
332 udp_rinitv4
333 udp_rinitv6
334 udp_valid_levels_arr
335 udp_winit
336 udpinfov4
337 udpinfov6
338 winit_arp
339 nxge_cksum_workaround
```